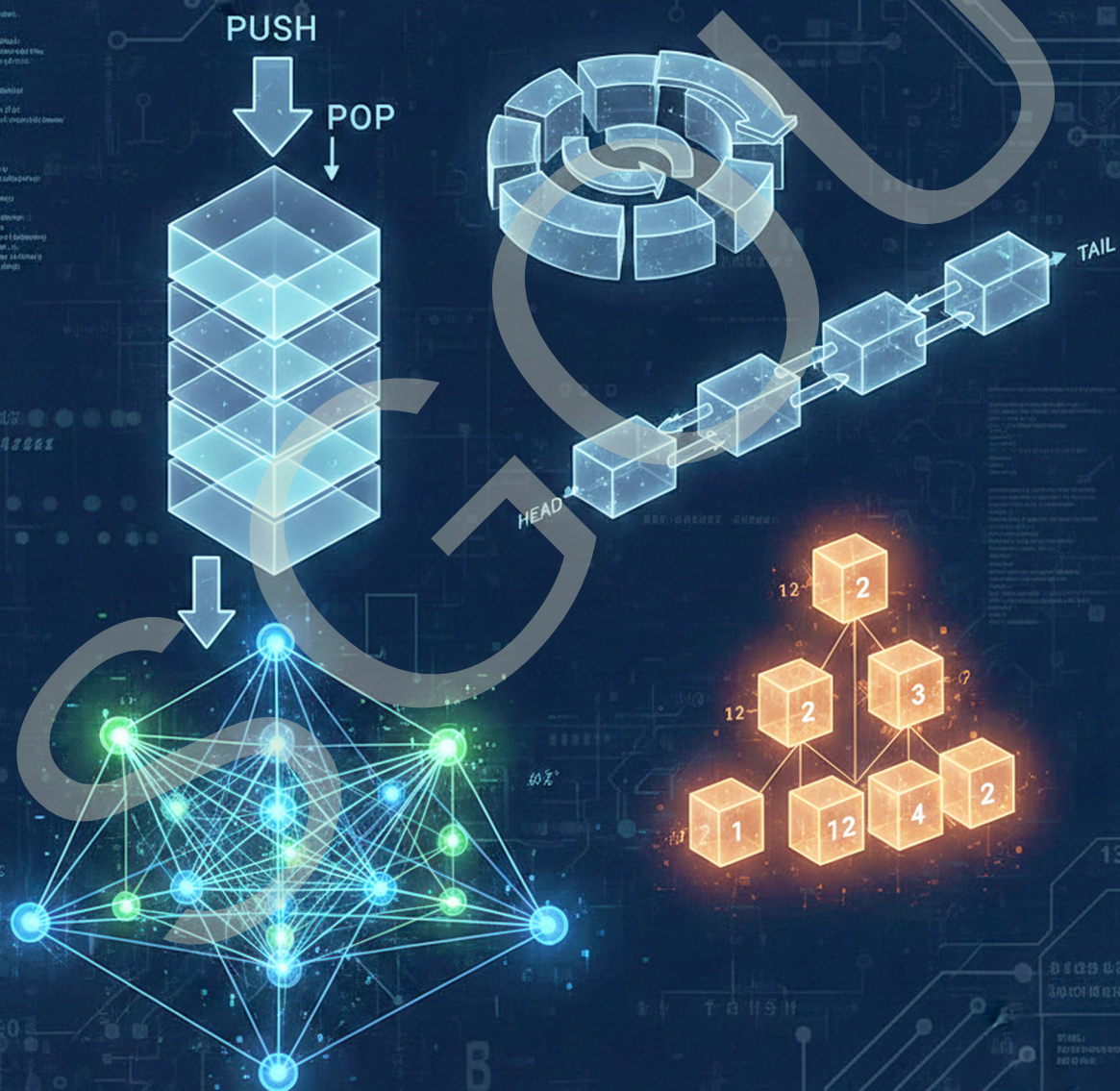


DATA STRUCTURES

Course Code: B24DS05DC
BSc Data Science and Analytics
Discipline Core Course
Self Learning Material



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Data Structures
Course Code: B24DS05DC
Semester - III

Discipline Core Course
Undergraduate Programme
BSc Data Science and Analytics
Self Learning Material
(With Model Question Paper Sets)



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala



DATA STRUCTURES

Course Code: B24DS05DC

Semester- III

Discipline Core Course

BSc Data Science and Analytics

Academic Committee

Dr. T. K. Manoj
Dr. Smitha Dharan,
Dr. Satheesh S.
Dr. Vinod Chandra S.S.
Dr. Hari V. S.
Dr. Sharon Susan Jacob
Dr. Ajith Kumar R.
Dr. Smiju I.S.
Dr. Nimitha Aboobaker

Development of the Content

Shamin S., Dr. Jennath H.S.,
Suramy Swamidas P.C.,
Greeshma P.P., Sreerekha V.K.,
Anjitha A.V., Dr. Kanitha Divakar,
Aswathy V.S., Subi Priya Laxmi S.B.N.,
Sheena C.V., Abhayadev M.,
Thafseela Koya, Unnikrishnan

Review and Edit

Dr. Sheeba K.

Linguistics

Dr. Sheeba K.

Scrutiny

Shamin S., Greeshma P.P.,
Sreerekha V.K., Anjitha A.V.,
Aswathy V.S., Dr. Kanitha Divakar,
Subi Priya Laxmi S.B.N.

Design Control

Azeem Babu T.A.

Cover Design

Jobin J.

Co-ordination

Director, MDDC :
Dr. I.G. Shibi
Asst. Director, MDDC :
Dr. Sajeevkumar G.
Coordinator, Development:
Dr. Anfal M.
Coordinator, Distribution:
Dr. Sanitha K.K.



Scan this QR Code for reading the SLM
on a digital device.

Edition
September 2025

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-994027-6-8



All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.in



Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities. The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.



Regards,
Dr. Jagathy Raj V. P.

01-09-2025

Contents

Block 01	Basics of C Programming for Data Structures	1
Unit 1	Fundamentals of C Programming	2
Unit 2	Pointers and Structures in C	75
Unit 3	Dynamic memory allocation in C	85
Block 02	Basic Data Structures	95
Unit 1	Introduction to Data Structures	96
Unit 2	Array as a Data Structure	111
Unit 3	Stack	125
Unit 4	Queue	150
Block 03	Linear Data Structures	169
Unit 1	Linked List Basics	170
Unit 2	Operations on Linked List	184
Unit 3	Linked List Representation of Stack and Queue	199
Unit 4	Linked List Types	214
Block 04	Non-Linear Data Structures	222
Unit 1	Trees	223
Unit 2	Binary Search Tree	241
Unit 3	Balanced Binary Tree	252
Unit 4	Graphs	291
Block 05	Design Algorithms	336
Unit 1	Complexity of Algorithms	337
Unit 2	Searching and Sorting	356
Unit 3	Divide and Conquer Algorithms: Minmax, Binary Search, Quicksort	373
Unit 4	Minimum Cost Spanning Trees	393
Block 06	Advanced Data Structures	415
Unit 1	Introduction to Advanced Data Structures	416
Unit 2	Introduction to Hashing	434
Unit 3	Heap Data Structure	442
Unit 4	Priority Queues	456
Model Question Paper Sets		466



Basics of C Programming for Data Structures



Unit 1

Fundamentals of C Programming

Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ explain the purpose and use of different characters in forming valid C syntax
- ◆ apply the rules of the character set to create valid identifiers and expressions in simple C programs
- ◆ identify and classify different types of tokens used in C, including keywords, identifiers, constants, operators, and special symbols
- ◆ recall different types of operators used in C such as arithmetic, relational, logical, assignment, size of operator, etc

Prerequisites

In today's technology-driven world, almost every digital system from mobile apps to embedded devices relies on programming. Learning the C programming language opens the door to understanding how software interacts with hardware, making it an ideal starting point for aspiring programmers and engineers. Before diving into advanced concepts, this unit introduces the essential building blocks of C: data types, variables, constants, and operators. Mastering these basics forms the foundation for writing efficient and error-free programs.

Learning the C programming language is essential for anyone who wants to build a strong foundation in computer science and software development. To fully benefit from this unit, learners should have a basic understanding of how computers store and process information. Familiarity with the idea of numbers, characters, and logical conditions will be helpful. This unit will demystify how different types of data are declared, stored, and manipulated using variables and constants in C. It will also explain how operators are used to perform calculations, make decisions, and assign values skills that are critical to programming in any language.

Are you curious about how a calculator performs operations, or how software makes decisions based on conditions? This unit will show you how these everyday actions are



powered by simple C code. With hands-on examples and clear explanations, you'll discover how to use arithmetic, relational, logical, and assignment operators effectively. By the end, you'll be ready to write your own basic C programs and take your first confident steps into the world of software development.



Key words

Tokens, Keywords, variables, valid identifiers, literals, constants, Operators, comments, Special symbols.

Discussion

1.1.1 Introduction to C Programming

C is a powerful, versatile, and widely-used general-purpose programming language. It is known for its simplicity, efficiency, and flexibility. One of its major strengths lies in being a structured programming language, which means that a C program is composed of multiple functional units or modules. Each module is responsible for a specific task, making the entire program easier to understand, develop, and manage. C is a high-level language that had a profound impact on modern programming. It was developed in the early 1970s by Dennis Ritchie at Bell Laboratories to build system software, specifically the UNIX operating system. Since then, C has gained widespread popularity due to its efficiency, portability, and flexibility.

C is a machine-independent language, which means C programs can be run on various hardware platforms with minimal or no modification. This portability makes it ideal for developing a wide range of software applications including operating systems (like Windows), database systems (such as Oracle), version control systems (like Git), and even interpreters for other programming languages (e.g., Python). The following

diagram shows the execution of a 'C' program

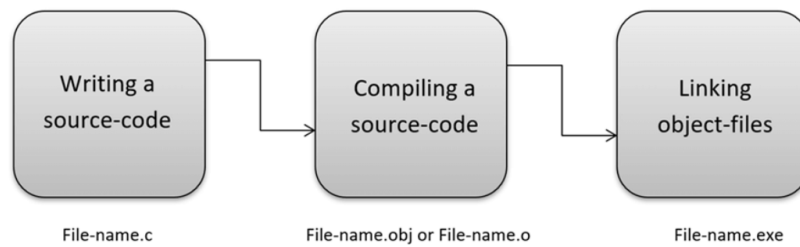


Fig1.1.1 Phases of Program Compilation in C

1.1.2 Features of C Programming language

- ◆ C has a simple and easy-to-understand syntax
- ◆ It supports structured programming using functions
- ◆ C programs execute fast as they are compiled into machine code
- ◆ It is portable and works across different computer systems
- ◆ C provides low-level access to memory through the use of pointers
- ◆ The language supports modular programming with reusable code blocks
- ◆ It includes a rich set of built-in functions and operators
- ◆ C allows dynamic memory management during program execution
- ◆ It is extensible and lets users define their own functions and data types
- ◆ C is widely used for developing operating systems, embedded systems, and system-level software

1.1.3 Character Set in C

In C programming, the **character set** refers to the collection of **valid characters** that the C compiler recognizes and uses to form programs. These characters are used to write identifiers, keywords, constants, operators, and other elements of the C language.

Table 1.1.1 Character set in C

Category	Examples	Description
1. Letters	A–Z, a–z	Uppercase and lowercase English alphabets
2. Digits	0–9	Decimal digits used in numeric constants and identifiers
3. Special Characters	`+ - * / = < > ! % &`	^ ~ ` etc.

- Valid: `_temp`, `Sum`, `data1`
- Invalid: `1value` (starts with a digit)
- ◆ Subsequent characters may include letters (A–Z or a–z), digits (0–9), or underscores
 - Valid: `value1`, `mark_10`
 - Invalid: `value#` (contains special character `#`)
- ◆ Keywords cannot be used as identifiers
 - C has a set of reserved words like `int`, `return`, `while`, etc., which have special meaning and cannot be redefined.
 - Invalid: `float int`; → Here, `int` is a keyword, not a valid identifier.
- ◆ Whitespace characters are not allowed
 - Identifiers must not contain spaces or tabs.
 - Invalid: `total marks` (should be `total_marks` or `totalMarks`)
- ◆ Identifiers are case-sensitive
 - `Total` and `total` are treated as two distinct identifiers.
- ◆ The name should be meaningful
 - While C allows any valid combination of characters within the rules, using descriptive names (e.g., `totalMarks` instead of `tm`) improves code readability and maintainability.
- ◆ There is a practical limit on length
 - Standard C (as per ANSI C) guarantees recognition of only the first 31 characters of an identifier (and 63 for external identifiers). While modern compilers often support longer names, it's advisable to keep them concise and within readable limits.

A variable in C is a named storage location in the computer's memory that holds a value which can be changed during the execution of a program.

Syntax:

`data_type variable_name;`

Examples

`int age;` `// Declaration`

`age = 25;` `// Assignment`

`float salary = 55000;` `// Declaration with initialization`

Data Types in C

In C programming, **data types define the type of data a variable can hold**. This ensures that memory is used efficiently and operations on data are performed correctly. C supports several built-in and user-defined data types to handle a variety of data formats such as integers, floating-point numbers, characters, and more.

Table 1.1.2 Data type in C

Category	Data Type	Size (Typical)	Format Specifier	Description
Basic (Primary)	int	2 or 4 bytes	%d or %i	Stores integers (positive or negative whole numbers).
	float	4 bytes	%f	Stores real numbers with decimal points (single precision)
	double	8 bytes	%lf	Stores real numbers (double precision)
	char	1 byte	%c	Stores a single character (ASCII)
Derived	array	Depends on elements	-	Collection of elements of the same Type
	pointer	2/4/8 bytes	x _p	Stores the memory address of another variable
	structure	Varies	-	Combines different data types into a single unit.
Void	Union	Varies	-	Similar to struct but uses shared memory for all members.
	function	-	-	A block of code that performs a specific task and returns a value.
Void	void	No size	-	Represents no value; used for functions that return nothing.
User-Defined	struct	Varies	-	Used to create custom data types combining multiple fields.

	union	Varies	-	Custom type like struct, but with shared memory for fields.
	enum	Usually 4 bytes	%d	Defines a set of named integer constants.
Modified Types	short int	2 bytes	%hd	Integer with smaller range
	long int	4 or 8 bytes	%ld	Integer with larger range.
	long long int	8 bytes	%lld	Very large integer values.
	unsigned int	2 or 4 bytes	%u	Integer that can only store positive values.
	unsigned char	1 byte	%c / %u	Stores positive ASCII values (0-255).
	long double	10/12/16 bytes	%Lf	Extended precision for real numbers.

1.1.4.3 Literals

A literal is a constant value directly written in the source code. These values do not change during the program's execution and are used to represent fixed data.

Table 1.1.3 Types of Literals in C

Name of literals	Explanation	Examples
Integer Literals	Represent whole numbers without decimal points.	<pre>int a = 100; // Decimal int b = 012; // Octal (starts with 0) int c = 0xFF; // Hexadecimal (starts with 0x)</pre>
Floating-Point Literals	Represent real numbers with decimal points or in exponential notation.	<pre>float x = 3.14; float y = 2.5e3; // $2.5 \times 10^3 = 2500.0$</pre>
Character Literals	Enclosed in single quotes, represent a single character.	<pre>char ch = 'A';</pre>
String Literals	String literals are sequences of characters enclosed in double quotes. They are stored as arrays of characters ending with a null character (<code>\0</code>).	<pre>char str[] = "Hello, World!";</pre>

1.1.4.4 Constants

Constants are fixed values that do not change during the execution of a program. They can be of different types such as integer constants, floating-point constants, character constants, or symbolic constants defined using `#define`.

Examples: 100, 3.14, 'A', `"#define PI 3.14"`

Literals and constants play a vital role in making C programs clear, reliable, and easy to maintain. **Literals** represent fixed values used directly in the code, while **constants** are named values that do not change during program execution. By using literals, programmers can express data values clearly, and by using constants, they can avoid accidental changes to important values and improve code readability.

1.1.4.5 Operators

Imagine you're building a calculator, writing a game, or developing software that makes decisions based on input. How does your program actually perform tasks like addition, comparison, or logic? That's where *operators* in C come into play. They act like tools in a toolbox, allowing you to manipulate data, evaluate conditions, and control the behavior of your program efficiently. Whether you're adding numbers, comparing values, or making decisions, operators are the unsung heroes behind the scenes of every powerful C program.

Operators in C are special symbols or keywords that instruct the compiler to perform specific operations on variables and values. These operations can be arithmetic, logical, relational, assignment-related, or more, and they form the core of most computations in a C program. **Operand** refers to the value or variable on which an operator performs an action. In an expression, operands are the data items being manipulated.

Example:

```
int a = 5, b = 10;  
int sum = a + b;
```

In example 1, **a** and **b** are **operands**, and **+** is the **operator**. The operator **+** performs addition on the operands **a** and **b**.

In C programming, operators are not one-size-fits-all, they come in various types, and each designed for a specific kind of task. Whether you're performing a simple addition, checking if two values are equal, or shifting bits in memory, C provides a rich set of operators to handle it. Understanding the different types of operators is essential for writing efficient, logical, and powerful code. Each type serves a unique purpose and helps you control how data is manipulated, compared, or assigned in your program. The C programming language has a large number of built-in operators, and they are classified into the following types:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Bitwise operators



5. Assignment operators
6. Conditional operators
7. Increment and decrement operators
8. Special operators

1. Arithmetic operators

Arithmetic operators are the most fundamental and commonly used operators in C programming. They allow you to perform basic mathematical operations such as addition, subtraction, multiplication, division, and finding the remainder. Whether you're calculating totals, measuring differences, or building mathematical logic into your program, arithmetic operators are essential tools that make numerical computation easy and efficient. Their simplicity and wide usage make them a crucial starting point for anyone learning to program in C. The C language provides the basic arithmetic operators, and they are listed in table 1.1.4.

Table 1.1.4 Arithmetic Operators

Operator	Operation	Description	Example
+	Addition	Adds the two numeric values on either side of the operator.	<pre>#include<stdio.h> #include<conio.h> void main() { int a=5, b=7; printf("the sum = %d, a+b); getch(); }</pre>
-	Subtraction	Subtracts the operand on the right from the operand on the left	<pre>#include<stdio.h> #include<conio.h> void main() { int a=5, b=7; printf("the difference between %d and %d = %d",b,a b-a); getch(); }</pre>

*	Multiplication	Multiplies the two values on both sides of the operator	<pre>#include<stdio.h> #include<conio.h> void main() { int a=5, b=7; printf("the product = %d, a*b); getch(); }</pre>
/	Division	Divides the operand on the left by the operand on the right and returns the quotient	<pre>#include<stdio.h> #include<conio.h> void main() { int a=9, b=2; printf("the quotient = %d, a/b); getch(); }</pre>
%	Modulus	Divides the operand on the left by the operand on the right and returns the remainder	<pre>#include<stdio.h> #include<conio.h> void main() { int a=9, b=2; printf("the reminder = %d, a%b); getch(); }</pre>

2. Relational operators

Relational operators in C are essential for making comparisons between values or expressions. They help your program decide the relationship between two data items whether one is greater than, less than, equal to, or different from another. These operators are the foundation of decision-making in programming, allowing you to control the flow of your code by evaluating conditions and executing different actions based on those comparisons. Understanding relational operators is key to writing programs that respond intelligently to different inputs and situations. The relational operators supported by C are listed in Table 1.1.5.

Table 1.1.5 Relational Operators

Operator	Operation	Description	Example
<code>==</code>	Equals to	If the values of two operands are equal, then the condition is True, otherwise, it is False.	<code>a == b</code>
<code>!=</code>	Not equal to	If the values of two operands are not equal, then the condition is True. Otherwise, it is False.	<code>a != b</code>
<code>></code>	Greater than	If the value of the left-side operand is greater than the value of the right side operand, then the condition is True, otherwise, it is False	<code>a > b</code>
<code><</code>	Less than	If the value of the left-side operand is less than the value of the right side operand, then the condition is True. Otherwise, it is False.	<code>a < b</code>
<code>>=</code>	Greater than or equal to	If the value of the left-side operand is greater than or equal to the value of the right-side operand, then the condition is True. Otherwise, it is False.	<code>a >= b</code>
<code><=</code>	Less than or equal to	If the value of the left operand is less than or equal to the value of the right operand, then it is True. Otherwise it is False	<code>a <= b</code>

3. Logical operators

Logical operators in C are used to make decisions based on multiple conditions. They allow a program to evaluate complex expressions and determine whether certain combinations of conditions are true or false. These operators are especially useful in control flow statements like `if`, `while`, and `for` loops, where decision-making is essential. Whether you want to ensure that two conditions are met, at least one is true, or to reverse a condition's result, logical operators help add intelligent behavior to your programs.

The logical operators used in C are listed in Table 1.1.6.

Table 1.1.6 Logical operators

Operator	Operation	Description	Example
&&	Logical AND	If both the operands are true, then the condition becomes True.	(a > b) && (b > c)
	Logical OR	If any of the two operands are True, then the condition becomes True.	(a > b) (b > c)
!	Logical NOT	Used to reverse the logical state of its operand	!(a == b)

5. Bitwise operators

Bitwise operators in C allow you to manipulate data at the individual bit level, giving you powerful control over how information is stored and processed. Unlike arithmetic or logical operators that work with whole values, bitwise operators perform operations directly on the binary representation of integers. These operators are essential in low-level programming, such as system programming, device drivers, and performance optimization, where efficiency and precise control over bits are crucial. Understanding bitwise operators opens up new possibilities for handling data in a highly efficient way. Data is manipulated at the bit level with bitwise operators. Shifting bits from right to left is often performed by these operators. Float and double data types do not support bitwise operators.

The & (bitwise AND)

The & (bitwise AND) in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. The output of bitwise AND is 1 if the corresponding bits of two operands are 1. If either bit of an operand is 0, the result of the corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

00001000 = 8 (In decimal)

Example

```
#include <stdio.h>

int main() {
    int a = 12, b = 25;
    printf("Result = %d", a&b);
    return 0;
}
```

Output

Result = 8

The | (bitwise OR)

The | (bitwise OR) in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

Let us suppose the bitwise OR operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

```
00001100
| 00011001
-----
00011101 = 29 (In decimal)
-----
```

Example

```
#include <stdio.h>

int main() {
    int a = 12, b = 25;
    printf("Result = %d", a | b);
    return 0;
}
```

Output

Result = 29

The ^ (bitwise XOR)

The ^ (bitwise XOR) in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

Let us suppose the bitwise XOR operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

```
00001100
^ 00011001
-----
00010101 = 21 (In decimal)
```

Example

```
#include <stdio.h>

int main() {
    int a = 12, b = 25;
    printf("Result = %d", a ^ b);
    return 0;
}
```

Output

Result = 21

The << (left shift)

The << (left shift) in C takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

Examples:

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 = 11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) = 3392 (In decimal)

The >> (right shift)

The >> (right shift) in C takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

Examples:

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

Example

```
#include <stdio.h>

int main ()
{
    int num = 212, i;
    for (i = 0; i <= 2; ++i)
    {
        printf ("Right shift by %d: %d\n", i, num >> i);
    }
    printf ("\n");

    for (i = 0; i <= 2; ++i)
    {
        printf ("Left shift by %d: %d\n", i, num << i);
    }
    return 0;
}
```

Output

Right shift by 0: 212

Right shift by 1: 106

Right shift by 2: 53

Left shift by 0: 212

Left shift by 1: 424

Left shift by 2: 848

The ~ (bitwise NOT)

The ~ (bitwise NOT) in C takes one number and inverts all bits of it.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

11011100 = 220 (In decimal)

Example

```
#include <stdio.h>

int main ()
{
    printf ("Output = %d\n", ~35);
    printf ("Output = %d\n", ~-12);
    return 0;
}
```

Output

Output = -36

Output = 11

Points to remember while using Bitwise operators

- ◆ The left shift and right shift operators should not be used for negative numbers.
- ◆ The bitwise OR of two numbers is just the sum of those two numbers if there is no carry involved, otherwise you just add their bitwise AND.
- ◆ The bitwise XOR operator is the most useful operator. It is used in many problems. A simple example could be “Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number” This problem can be efficiently solved by just doing XOR to all numbers.
- ◆ The bitwise operators should not be used in place of logical operators.
- ◆ The & operator can be used to quickly check if a number is odd or even. The value of the expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

- ◆ The \sim operator should be used carefully. The result of the \sim operator on a small number can be a big number if the result is stored in an unsigned variable. And the result may be a negative number if the result is stored in a signed variable (assuming that the negative numbers are stored in 2's complement form where the leftmost bit is the sign bit)
- ◆ The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively. As mentioned in point 1, it works only if numbers are positive.

Table 1.1.7 Bitwise operators

Operator	Operation	Truth Table				
&	Bitwise AND	a	b	a&b	a b	a^b
	Bitwise OR	0	0	0	0	0
^	Bitwise exclusive OR	0	1	0	1	1
<<	left shift	1	0	0	1	1
>>	right shift	1	1	1	1	0

The bitwise shift operator changes the value of a bit. The left operand specifies the value to be moved, while the right operand specifies the number of places the value's bits must be shifted. The precedence of both operands is the same.

Example:

a = 0001000

b = 2

a << b = 0100000

a >> b = 0000010

6. Assignment operators

Assignment operators in C are used to store values into variables, making them fundamental to programming. The simplest assignment operator is the equal sign (=), which assigns the value on the right to the variable on the left. Beyond this basic form, C offers compound assignment operators like +=, -=, *=, and /=, which combine arithmetic operations with assignment in a concise way. These operators help write

cleaner and more efficient code by reducing redundancy while updating variable values.

Table 1.1.8 Assignment Operators

Operator	Description
=	Assigns value from right-side operand to left side variable
+=	It adds the value of the right-side operand to the left-side operand and assigns the result to the left-side operand Note: $x += y$ is the same as $x = x + y$
-=	It subtracts the value of the right-side operand from the left-side operand and assigns the result to the left-side operand Note: $x -= y$ is the same as $x = x - y$
*=	It multiplies the value of the right-side operand with the value of the left-side operand and assigns the result to the left-side operand Note: $x *= y$ is the same as $x = x * y$
/=	It divides the value of the left-side operand by the value of the right-side operand and assigns the result to the left-side operand Note: $x /= y$ is the same as $x = x / y$
%=	It performs modulus operation using two operands and assigns the result to left-side operand Note: $x \% = y$ is the same as $x = x \% y$

Let us consider a simple C program snippet to understand the assignment operators

```
int a = 21;

int c ;

c = a;
printf("Line 1 - = Operator Example, Value of c = %d\n", c );

c += a;
printf("Line 2 - += Operator Example, Value of c = %d\n", c );

c -= a;
printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

c *= a;
printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

c /= a;
printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

c = 200;
```

```
c %= a;  
printf("Line 6 - %= Operator Example, Value of c = %d\n", c );
```

Output:

```
Line 1 - = Operator Example, Value of c = 21  
Line 2 - += Operator Example, Value of c = 42  
Line 3 - -= Operator Example, Value of c = 21  
Line 4 - *= Operator Example, Value of c = 441  
Line 5 - /= Operator Example, Value of c = 21  
Line 6 - %= Operator Example, Value of c = 11
```

7. Conditional operators

The conditional operator, also known as the ternary operator, is a concise way to perform decision-making in C. It allows you to evaluate a condition and choose between two values or expressions in a single line of code, making your programs shorter and often easier to read. It needs three operands to work on. The conditional operator is a powerful shortcut for simple **if-else** statements, helping you write clean and efficient code. The syntax of the ternary operator is:

`expression 1 ? expression 2 : expression 3`

- ◆ The question mark “?” in the syntax to check the condition.
- ◆ The first expression (expression 1) usually returns true or false, which determines whether (expression 2) will be executed or otherwise (expression 3).
- ◆ If (expression 1) is valid, the expression on the left side of “:”, i.e. (expression 2), is executed.
- ◆ If expression 1 returns false then expression 3 will be executed.

For example, consider the following statements.

```
v1 = 5;  
v2 = 20;  
Result = v1 > v2 ? v1 : v2;
```

The result will be assigned the value of v2.

8. Increment and decrement operators

Increment (++) and decrement (--) operators in C provide a quick and convenient

way to increase or decrease the value of a variable by one. They are widely used in loops, counters, and iterative processes to simplify code and improve readability. These operators come in two forms , prefix (e.g., `++a`) and postfix (e.g., `a++`), which differ slightly in how they change the variable's value during expressions. Mastering these operators is essential for writing efficient and clean C programs.

Table 1.1.9 Increment or Decrement operators

Operator	Operation	Description	Example
++	Increment	Operates on a single value Increases the value of the integer operand by 1	<pre>#include<stdio.h> #include<conio.h> void main() { int a=9, b; b = ++a; printf("the output of increment operator = %d", b); getch(); }</pre>
--	Decrement	Operates on a single value Decreases of the value of the integer operand by 1	<pre>#include<stdio.h> #include<conio.h> void main() { int a=9, b; b = --a; printf("the output of increment operator = %d", b); getch(); }</pre>

Rules for increment and decrement operator:

1. Increment and decrement operators are unary operators.
2. They need variables as their operand.
3. When a variable is used in an expression with a postfix increment or decrement, the expression is evaluated first using the initial value of the variable, then update the value of the variable by one.
4. When the prefix increment or decrement is used with a variable in an expression, the value of the variable is updated first, and then the expression is evaluated.

Program:

```
int main()
{
    int x = 10, a, y = 10, b;
    a = ++x;
    b = y++;
    printf("Pre Increment Operation");
    // Value of a will change
    printf( "\na = %d ", a);
    // Value of x changes before execution of a=++x;
    printf( "\nx = %d", x);
    printf("Post Increment Operation");
    // Value of b will change
    printf( "\nb = %d ", b);
    // Value of y changes before execution of b=y++;
    printf( "\ny = %d", y);
    return 0;
}
```

Output

Pre Increment Operation

a = 11

x = 11

Post Increment Operation

b = 10

y = 11

In the above example, we have used the increment operator. The decrement operator will work in the same pattern.

Special operators

Special operators in C include a variety of unique tools that provide additional capabilities beyond basic arithmetic and logical operations. These operators handle specific tasks such as determining the size of data types (**sizeof**) and working with pointers (**& for address-of, * for dereferencing**). Understanding these special operators is important for mastering advanced programming concepts like memory management, efficient coding, and complex expression evaluation. The C language supports some special operators who are listed in table 1.1.10.

Table 1.1.10 Special operators

Operator	Description	Example
sizeof	Returns the size of a variable	sizeof(x) return size of the variable x
&	Returns the address of a variable	&x; return address of the variable x
*	Pointer to a variable	*x; will be a pointer to a variable x

Sample code:

1. *sizeof* Operator

Example

```
#include <stdio.h>

int main() {
    int a;

    printf("Size of int: %zu bytes\n", sizeof(a));
    printf("Size of double: %zu bytes\n", sizeof(double));

    return 0;
}
```

Output

Size of int: 4 bytes

Size of double: 8 bytes

2. Address-of Operator (&)

Example

```
#include <stdio.h>

int main() {
    int a = 10;

    printf("Address of a: %p\n", &a);

    return 0;
}
```

Output

Address of a: 0x7ffeebfff5ac

3. Dereference Operator (*)

Example

```
#include <stdio.h>

int main() {
    int a = 10;

    int *ptr = &a;

    printf("Value of a through pointer: %d\n", *ptr);

    return 0;
}
```

Output

Value of a through pointer: 10

Operators in C are fundamental building blocks that enable programmers to perform a wide range of tasks, from simple arithmetic to complex logical decisions. This unit introduced various types of operators, including **arithmetic**, **relational**, **logical**, **bitwise**, **assignment**, **conditional**, **increment/decrement**, and **special operators**, each serving a specific purpose in manipulating data and controlling program flow. Understanding how and when to use these operators is essential for writing efficient, readable, and effective C programs. Mastery of operators forms the foundation for developing logical expressions, performing calculations, and managing memory and control structures in C programming.

1.1.4.6 Special Symbols

Special symbols are characters that serve specific purposes in C syntax. They are used

in defining blocks, statements, arrays, and more.
Examples:

- ◆ {} – Used to define a block of code
- ◆ () – Used in functions and conditions
- ◆ [] – Used in arrays
- ◆ ; – Statement terminator
- ◆ # – Used for preprocessor directives
- ◆ “ – Used to enclose strings
- ◆ ‘ – Used to enclose single characters

1.1.5 Comments in C Programming

In C programming, comments are non-executable parts of code used to explain the logic, make the code more readable, or temporarily disable code for testing or debugging. The compiler ignores comments, so they do not affect the actual execution of the program.

Table 1.1.11 Types of Comments in C

Type	Syntax	Description
Single-line	// comment	Used to write brief comments on a single line.
Multi-line	/* comment block */	Used to write longer comments spanning multiple lines.

Recap

- ◆ C is a structured and procedural programming language that serves as the foundation for many modern languages.
- ◆ Data types in C define the type of data a variable can store, such as int, float, char, and double.
- ◆ Variables are named memory locations used to store data, and they must be declared before use.
- ◆ Constants are fixed values that cannot be changed during the execution of a program, declared using the const keyword.
- ◆ Literals are actual constant values written directly in the code, such as 10, 'A', or 3.14.
- ◆ Comments in C help document the code and are ignored by the compiler. Single-line comments use //, and multi-line comments use /* ... */.

- ◆ Arithmetic operators perform basic mathematical operations like addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).
- ◆ Relational operators compare values and return a boolean result: equal (==), not equal (!=), greater than (>), less than (<), etc.
- ◆ Logical operators like && (AND), || (OR), and ! (NOT) are used to combine or invert conditional expressions.
- ◆ Increment (++) and decrement (--) operators are used to increase or decrease a variable's value by one, either before or after evaluation.
- ◆ Conditional (ternary) operator ?: provides a shorthand way to choose between two expressions based on a condition.
- ◆ Assignment operators like =, +=, -=, *=, /=, and %= are used to assign and update values of variables.
- ◆ Special operators in C include the pointer *, address-of &, and sizeof, which returns the size of a variable or data type in bytes.
- ◆ The sizeof operator is a compile-time operator that helps in memory management by determining the size of data types or variables.

Objective Type Questions

1. Which of the following is a valid C data type?
2. What keyword is used to define a constant in C?
3. Which operator is used to assign a value to a variable?
4. What will the expression `10 % 3` return?
5. What is the result of `5 > 2 && 3 < 4`?
6. Which operator checks if two values are equal?
7. What is the size (in bytes) of `int` in most systems?
8. Which of the following is a correct variable name in C?
9. What is the purpose of comments in a C program?
10. What symbol is used for single-line comments in C?

11. What is a literal in C?
12. What does `sizeof(float)` return (in most systems)?
13. Which operator has the highest precedence in C?
14. What is the output of `printf("%d", 3 == 4);`?
15. Which operator is used to increase a variable's value by 1?
16. What is the result of `!(1 && 0)`?
17. How is a multi-line comment written in C?
18. Which of the following is a character literal?
19. The expression `a ? b : c` is an example of which operator?
20. Which operator gives the memory size of a variable or type?
21. What does the `&` operator return in C?
22. What is the output of `int a = 5; printf("%d", ++a);`?
23. Which of the following is a floating-point literal?
24. What will `a += 2;` do if `a = 5;`?

Answers to Objective Type Questions

1. `int`
2. `const`
3. `=`
4. `1`
5. `1 (true)`
6. `==`
7. `4`
8. `total_sum`

9. To improve code readability
10. //
11. A fixed value written directly in code (e.g., 5, 'A')
12. 4
13. () (parentheses)
14. 0
15. ++
16. 1
17. /* comment */
18. 'B'
19. Conditional (ternary) operator
20. sizeof
21. Address of a variable
22. 6
23. 3.14
24. It will update a to 7

Assignments

1. Write a C program that demonstrates the use of all basic data types (**int**, **float**, **char**, **double**) along with variables, constants, and literals.
2. Create a C program that accepts two integers from the user and demonstrates the use of arithmetic, relational, and logical operators.
3. Explain with examples the difference between pre-increment, post-increment, pre-decrement, and post-decrement operators in C.
4. Design a menu-driven C program using the conditional (ternary) operator and assignment operators to perform simple calculations like addition, subtraction, multiplication, and division.

Reference

1. Kernighan, B. W., & Ritchie, D. M. (2023). *The C Programming Language* (2nd ed.). Pearson Education.
2. Griffiths, D., & Griffiths, L. (2022). *Head First C: A Brain-Friendly Guide*. O'Reilly Media.
3. Oualline, S. (2021). *Practical C Programming* (4th ed.). O'Reilly Media.
4. Schildt, H. (2020). *C: The Complete Reference* (4th ed.). McGraw-Hill Education.
5. E. Balagurusamy. (2019). *Programming in ANSI C* (8th ed.). McGraw Hill Education.

Suggested Reading

1. GeeksforGeeks <https://www.geeksforgeeks.org/c-programming-language/>
2. TutorialsPoint – C Programming <https://www.tutorialspoint.com/cprogramming/index.html>
3. Programiz – Learn C Programming <https://www.programiz.com/c-programming>
4. Cplusplus.com (C Language Reference) <https://cplusplus.com/reference/library/>

Unit 2

Control Flow and Arrays in C

Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ define various control structures in C, including conditional statements, looping statements, arrays, and functions.
- ◆ List different types of conditional statements, looping constructs, and function types used in C programming.
- ◆ Describe the purpose and working of if, if-else, switch, for, while, do-while, and user-defined functions.
- ◆ Explain how arrays and functions are declared, defined, and used in a C program.
- ◆ Classify functions into categories such as with/without arguments and with/without return values.

Prerequisites

Imagine using a washing machine. You start it, select a mode, and it automatically follows a sequence such as soaking, washing, rinsing, and spinning. Each stage runs based on a condition, repeats actions for a set time, and then moves to the next. This is similar to how control statements, loops, arrays, and functions work in C programming. Control statements like if or switch help the program decide what action to take, just like the machine choosing between wash modes. Loops allow repeated tasks such as rotating the drum several times during a wash. Arrays are used to store multiple values, like water levels or spin speeds for different programs. Functions break the overall process into smaller tasks, where one function may handle washing, another rinsing, and so on. Without these elements, the washing machine or a C program would be difficult to manage or operate smoothly. In programming, using loops avoids writing the same code again and again, and functions improve clarity and reuse. Arrays keep related data organized, and control statements manage logical decisions. Just like the washing machine needs all these steps for a complete cycle, a C program needs these concepts to run correctly and efficiently.



Key words

If, switch, for, while, break, continue, return

Discussion

1.2.1 Conditional Statements

In everyday life, we often face situations where we must choose between available options. For instance, imagine you're about to make a cup of tea and check if there's any milk in the fridge.

You might ask yourself, "Is there milk available?"

If the answer is no, you'll need to add milk to your grocery list.

If the answer is yes, you'll go ahead and make the tea.

This illustrates a decision-making process, selecting a course of action based on a condition. Similarly, in programming, selection involves choosing between two or more paths depending on specific criteria. This section explains the selection control structures offered in C programming.

1.2.1.1 Simple if statement

The **if** statement is one of the most powerful tools for making decisions in programming. It plays a key role in directing the flow of a program based on specific conditions. Typically, an **if** statement includes a condition that is checked before any code inside its block is executed. Only if the condition evaluates to true will the statements within the block run. The general syntax of the **if** statement is shown below:

The syntax of the if statement for single-line statement is :

```
if(condition)
    Statement ;
```

The syntax of if statement for multi-line statements is :

```
if(condition)
{
    Statement1 ;
    Statement2 ;
    .
    .
    .
    StatementN ;
}
```

Suppose we want to display the positive difference between two numbers, num1 and num2. To achieve this, we need to modify our approach.

Take a look at Program below. It calculates the difference by subtracting the smaller number from the larger one, ensuring the result is always positive. The decision is made based on the values of num1 and num2.

```
/* program to print the positive difference between two numbers*/  
  
#include <stdio.h>  
  
#include <conio.h>  
  
void main()  
{  
    int num1, num2, diff;  
    printf("Enter the first number\t:\t");  
    scanf("%d",&num1);  
    printf("Enter the second number\t:\t");  
    scanf("%d", & num2);  
    if(num1 > num2)  
        diff = num1 - num2  
    if(num2 > num1)  
        diff = num2 - num1  
    printf("The positive difference between %d and %d is %d",num1, num2, diff);  
    getch();  
}
```

Output:

Enter the first number : 5

Enter the second number : 9

The positive difference between 5 and 9 is 4

In the above Program, two integer variables, num1 and num2, are declared and their values are taken as input from the user. The program then identifies the larger of the two numbers and computes the difference between them.

Let's consider another program:

```
/* program to check voting eligibility*/  
  
#include <stdio.h>  
  
#include <conio.h>  
  
void main()  
{  
  
    int age;  
  
    printf("Enter your age\t:\t");  
  
    scanf("%d", &age);  
  
    if(age >= 18)  
        printf("Eligible for voting");  
  
    if (age<18)  
        printf("Not eligible for voting");  
  
    getch();  
}
```

Output:

Enter your age : 45

Eligible for voting

Enter your age : 12

Not eligible for voting

1.2.1.2 if-else statement

The if-else statement is a variation of the if statement which allows one to write two alternate paths. The control condition will determine which path should be taken. The if-else sentence has the following syntax.

The syntax of if statement for single-line statement is :

```
if(condition)
    Statement ;

else
    Statement ;
```

The syntax of if statement or multi-line statements is :

```
if(condition)
{
    Statement1 ;
    Statement2 ;
    .
    .
    .
    StatementN ;
}

else
{
    Statement1 ;
    Statement2 ;
    .
    .
    .
    StatementN ;
}
```

Let us consider the flowchart to understand program flow in if-else statement

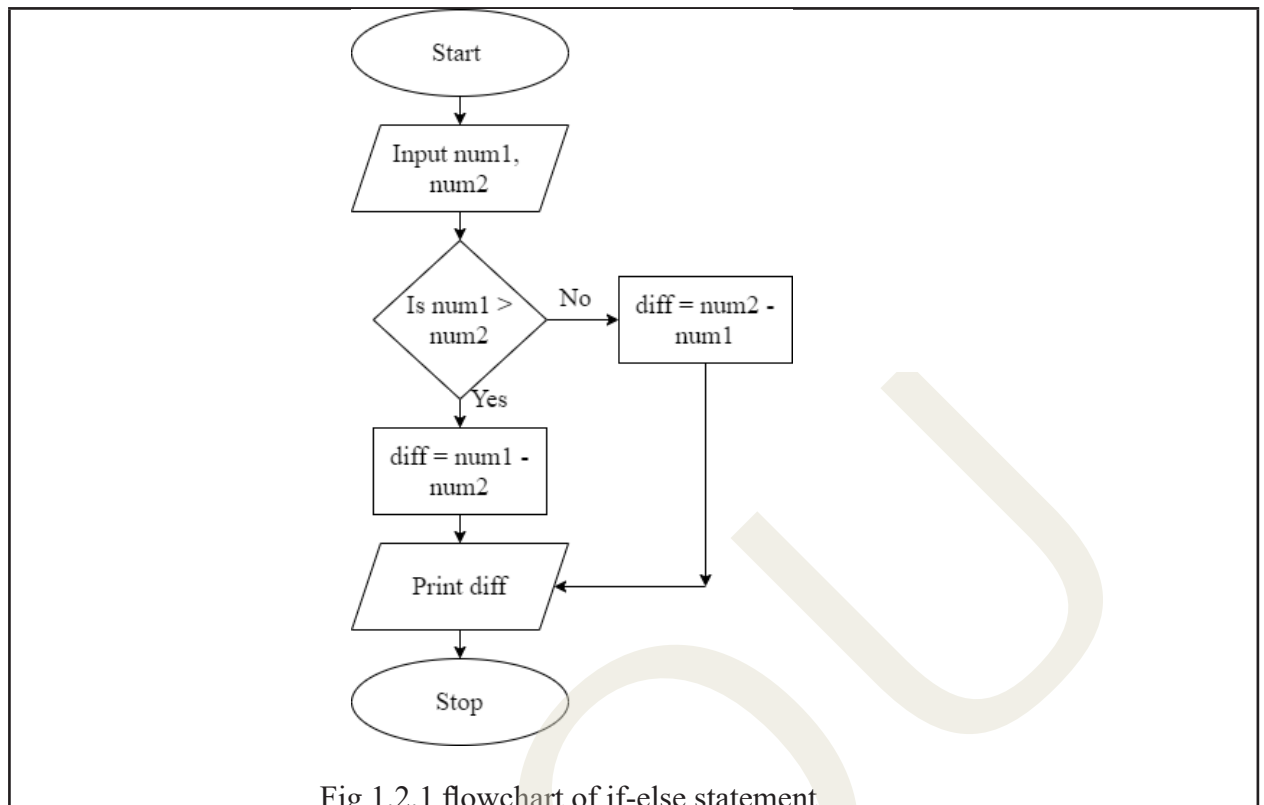


Fig 1.2.1 flowchart of if-else statement

Let us modify the above program with an if-else statement

```

/* program to check voting eligibility*/
#include <stdio.h>
#include <conio.h>
void main()
{
    int age;
    printf("Enter your age\t:\t");
    scanf("%d", &age);
    if(age >= 18)
        printf("Eligible for voting");
    else
        printf("Not eligible for voting");
    getch();
}
  
```

1.2.1.3 Nesting of if-else statement

Nested if-else statements are used when multiple levels of decision-making are needed. Placing one if-else block inside another is called nesting. Let's create a program to show how nested if-else structures work in practice.

```
#include<stdio.h>

#include<conio.h>

void main()
{
    int num=1;
    if(num<10)
    {
        if(num==1)
        {
            printf("The value is:%d\n",num);
        }
        else
        {
            printf("The value is greater than 1");
        }
    }
    else
    {
        printf("The value is greater than 10");
    }
    getch();
}
```

Output

The value is: 1

1. The code above demonstrates the use of nested if-else statements to determine

whether a number is less than or greater than 10 and display the result.

2. Initially, a variable num is assigned the value 1. An if-else structure is then used. The outer if condition checks whether the number is less than 10. If this condition is true, the program proceeds to the inner block.
3. Within the inner block, another condition checks if num is equal to 1. If this is true, the corresponding if block runs; otherwise, the else block executes. In this case, since the condition is satisfied, the if block executes and displays the output accordingly.

1.2.1.4 The if-else-if ladder

There are situations where multiple conditions need to be checked, resulting in several possible outcomes. In such cases, the **if-else if** structure is useful for linking these conditions together. The syntax for the **if-else if** construct is given below:

```
if( condition 1)
{
    statements;
}
else if( condition 2)
{
    statements;
}
else if(condition N)
{
    statements;
}
else
{
    statements;
}
```

The number of else if statements depends on how many conditions need to be checked.

If the initial condition is false, the next one is evaluated, and this continues until a true condition is found. When a valid condition is encountered, the associated block of code is executed, and the entire if structure ends.

To better understand the if-else if concept, let's look at a sample C program.

```
#include<stdio.h>

int main()
{
    int marks=83;
    if(marks>75){
        printf("First class");
    }
    else if(marks>65){
        printf("Second class");
    }
    else if(marks>55){
        printf("Third class");
    }
    else{
        printf("Fourth class");
    }
    return 0;
}
```

In this program, we begin by initializing a variable named mark. Several conditions are defined using the else-if ladder structure. The value of mark is first checked against the initial condition; if it matches, the corresponding message is displayed on the screen. If the first condition is false, the program moves on to evaluate the next one. This continues until a matching condition is found or all conditions are checked. If none are true, the control moves to the default statement, which is then executed.

1.2.1.5 The switch-case statement

In everyday situations, we often encounter scenarios where we have to choose from several alternatives instead of just two. For instance, selecting a school, picking a hotel, or making a career choice can involve multiple options. Similarly, in C programming, some decisions go beyond simple yes or no choices. To handle such cases efficiently, C offers a special control structure.

The switch statement, more precisely the switch case default construct, allows a program to choose from multiple possible actions. The general syntax of the switch case is presented below.

```
switch( expression )
{
    case value-1:
        statement(s);
        break;
    case value-2:
        statement(s);
        break;
    case value-n:
        statement(s);
        break;
    default:
        statement(s);
        break;
}
```

Rules for using the switch statement in C:

- ◆ The expression used in a switch must be either an integer or a character type.
- ◆ Values like 1, 2, and n act as case labels, each identifying a different case. It's important that no two case labels have the same value, as this would create issues during program execution.
- ◆ Each case label must be followed by a colon (:), and every case is linked to a specific block of code.

- ◆ The break statement is used at the end of each case to prevent the execution from continuing into the next case. Without a break, the program will keep executing subsequent cases until it reaches the end of the switch block.
- ◆ The default case is optional. It runs only if none of the defined case values match the expression. If all possible cases are already handled, the default case may be omitted.

The control flow that happens in the switch case is shown below

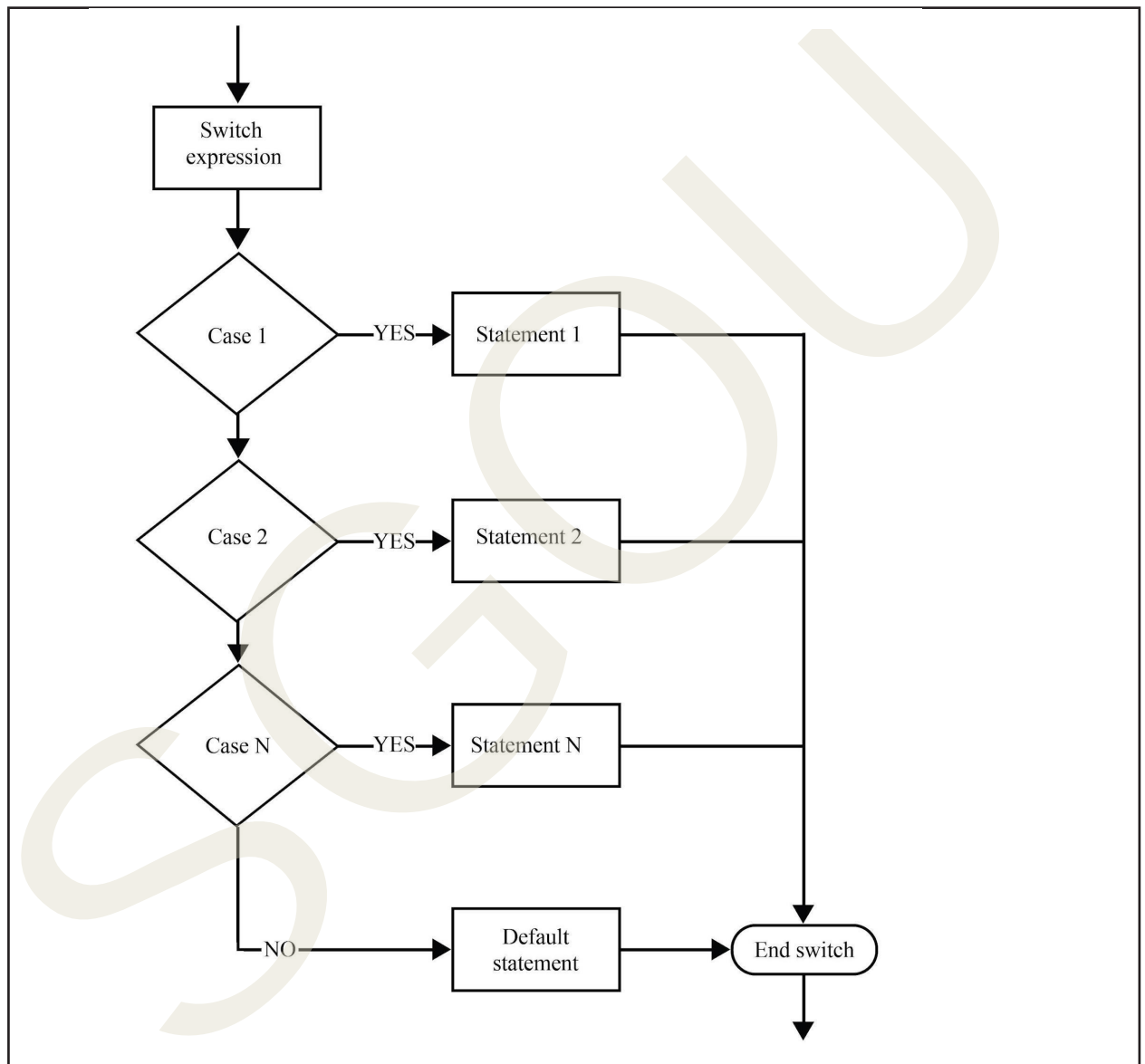


Fig. 1.2.2 Flowchart representation of Switch Statement

Let us understand the switch-case using an example

```

//Print day in a week
#include <stdio.h>
#include <conio.h>
void main() {
    int day_no;
    printf("Enter a Day Number\t:\t");
    scanf("%d", &day_no);
    switch (day_no) {
        case 1:
            printf("Sunday");
            break;
        case 2:
            printf("Monday");
            break;
        case 3:
            printf("Tuesday");
            break;
        case 4:
            printf("Wednesday");
            break;
        case 5:
            printf("Thursday");
            break;
        case 6:
            printf("Friday");
            break;
        case 7:
            printf("Saturday");
            break;
        default:
            printf("Enter a valid day number from 1 to 7");
            break;
    }
    getch();
}

```

Output

Enter a Day Number : 1

Sunday

Enter a Day Number : 12

Enter a valid day number from 1 to 7

Let us understand the program:

The switch statement is used to compare the value stored in the variable `day_no`, and the corresponding block of code for the matching case is executed. In this program, the value assigned to `day_no` is 1, so the switch executes the case labeled 1. If the input does not match any of the defined cases, the program proceeds to the default case. Once a matching case (or the default) has been executed, the control exits the switch block, and the output is displayed, completing the program successfully.

1.2.2 Loops

There are situations where certain actions need to be repeated. For example, consider hammering a nail. Without consciously thinking about it, you keep asking yourself, “Is the nail in?” If the answer is “no,” you strike it again. This process continues until the answer becomes “yes,” at which point you stop. This type of repeated action is known as a loop or iteration. Loops help programmers write efficient code by handling repetitive tasks without rewriting the same instructions multiple times.

Let us consider an example program to understand the loop concept.

```
/ A simple program to display the first five even numbers /
```

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
void main()
```

```
{
```

```
    printf("0\n");
```

```
    printf("2\n");
```

```
    printf("4\n");
```

```
    printf("6\n");
```

```
    printf("8\n");
```

```
}
```

This program uses a series of `printf` statements to display the first five even numbers starting from 0. Each number is printed on a new line.

What if we needed to print 1,000 or even 100,000 even numbers? Writing thousands of `printf` statements would be highly inefficient and impractical. Instead, the task can be

handled much more effectively using a loop or iteration.

Here's a more efficient approach using a basic algorithm:

1. Declare a variable, for example, count.
2. Initialize count with the value 0.
3. Check if count is divisible by 2 (i.e., remainder is 0).
4. If the condition is true, print the value of count.
5. Increase the value of count by 1.
6. Repeat steps 3 to 5 until count reaches 100,000.

Looping structures in programming are used to execute a set of instructions repeatedly, based on a given condition. These statements will continue to run as long as the condition remains true. The condition is typically checked using a variable known as the loop control variable. Once the condition becomes false, the loop stops executing. It is crucial for the programmer to design the loop in a way that ensures the condition eventually fails; otherwise, the loop would run infinitely. For instance, if there is no condition like $\text{count} \leq 100000$, the program could potentially crash. In C, there are three primary loop structures used for this purpose: for, while, and do-while.

1.2.2.1 for loop

The for loop is used to repeat a set of instructions for a specific range or sequence of values. Each item in that range is processed one at a time through the loop. These values can be numbers or elements from data types like strings or arrays, which will be covered in future chapters.

With each cycle of the loop, the control variable checks whether all the values in the range have been processed. The loop continues executing the statements inside it until all elements are covered. Once the loop completes, control moves to the instruction that follows the for loop. When using a for loop, the number of iterations is typically known beforehand. A flowchart (see Fig. 2.2.3) illustrates how a for loop operates.

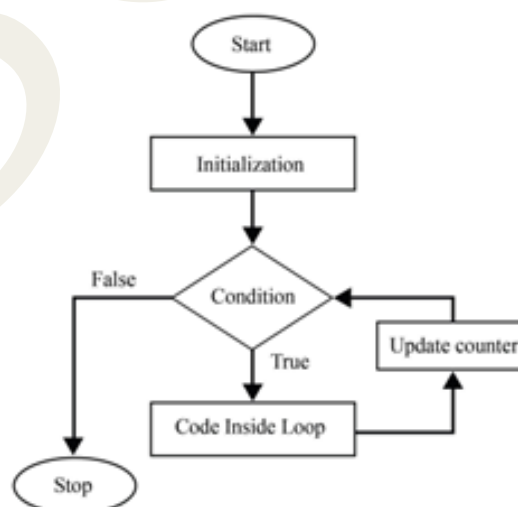


Fig. 1.2.3 Flowchart of a Loop Execution

The syntax of for loop is

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of the loop
}
```

- ◆ The initialStatement in a for loop is executed just once at the beginning.
- ◆ The testExpression is a condition that evaluates to either true or false; it checks the loop counter against a specified limit after every iteration and stops the loop when it evaluates to false.
- ◆ The updateStatement adjusts the counter value - either incrementing or decrementing it by a defined amount after each loop cycle.

Let us consider a C program to understand the **for** loop better

#include<stdio.h> int main() { int number; for(number=1;number<=10;number++) //for loop to print 1-10 nos { printf("%d\n",number); //to print the number } return 0; }	Output 1 2 3 4 5 6 7 8 9 10
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------

The above program prints the numbers from 1 to 10 using a for loop. Let us understand how it works?

1. An integer variable was declared to hold the values.
2. In the initialization part of the for loop, the variable number was set to 1. The condition for execution and the increment logic were defined in their respective sections of the loop.
3. Inside the loop's body, each number is displayed on a new line using the print function. After each iteration, the value of number increases by one, continuing the process until it reaches 10.

1.2.2.2 while loop

Earlier, we discussed how a **for loop** runs a set number of times. But what happens when we do not know in advance how many times the statements should repeat?

Take music streaming apps as an example. When you enable the repeat feature, your favorite song plays continuously until you manually stop or skip it.

In a similar way, the **while loop** repeatedly executes a block of code as long as its condition remains true. The condition is checked before each repetition, including the very first one. If the condition is false from the beginning, the loop's body will not execute at all.

The condition is evaluated before every cycle, and the loop continues as long as it remains true. Once it becomes false, the loop stops, and the control moves to the next instruction after the loop.

It is important that the logic inside the while loop eventually makes the condition false. Otherwise, the loop will continue forever, which causes a logical error in the program.

The syntax of while loop is:

```
while(condition)
{
    statement(s);
}
```

Let us consider a C program

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int num=1;    // initializing the variable
    while(num<=10) //while loop with condition
    {
        printf("%d ",num);
        num++; //incrementing operation
    }
    return 0;
}
```

Output

1 2 3 4 5 6 7 8 9 10

1.2.2.3 do-while loop

Previously, we learned that a **for loop** runs a specific number of times. But what should we do if we are unsure how many times a block of statements needs to be repeated?

Consider music streaming applications. When the repeat option is turned on, a song plays continuously until you decide to stop or switch to another track.

In the same way, a **while loop** keeps running a section of code as long as a certain condition remains true. This condition is checked before every execution, even before the first time. If the condition is false from the start, the loop's code will not run at all.

The loop continues to run as long as the condition stays true. When it becomes false, the loop ends and the program moves to the statement that follows it.

It is essential that the logic inside the loop eventually causes the condition to become false. If not, the loop will keep running endlessly, resulting in a logical error in the program, as demonstrated in Program.

The syntax of the do-while loop is:

```
do
{
    statement(s);
} while( condition );
```

Let us consider a C program to understand the concept clearly

```
/* program to print the first n even numbers*/

#include<stdio.h>

int main()
{
    int num=1, limit; //initializing the variable
    printf("Enter the limit: ");
    scanf("%d",&limit);
    do //do-while loop
    {
        if(num%2==0)
        {
```

```

        printf("%d ", num);
    }

    num++; //incrementing operation
} while(num<=limit);

return 0;
}

```

Enter the limit: 10

2 4 6 8 10

In the above program, we display the even numbers first up to the limit specified by the user. So, how does the program generate the sequence?

1. We start by initializing a variable `num` with the value 1 and declaring another variable `limit` to store the user's input.
2. The `scanf()` function is used to receive the input and assign it to `limit`. Next, we use a **do while loop** to control the process.
3. Inside the loop, we check if `num` divided by 2 leaves a remainder that is **not zero**, this confirms the number is odd.
4. If the condition is met, we print the value of `num`.
5. After each loop iteration, `num` is increased by 1.
6. This process continues until `num` reaches the value of `limit`.
7. Once that happens, the loop stops, and the statement following the loop, `return 0` in this case is executed.

1.2.2.4 break, continue and goto statements

The **break** statement is used to exit a loop immediately. When the program encounters a **break** inside a loop or a block of code, it stops executing that block and transfers control to the statement following it. This effectively ends the loop. The **break** statement is also commonly used in **switch case** structures to exit from a specific case.

Syntax:

```
break;
```

Let us understand the control flow of **break** command using a flow chart (refer Fig 2.2.4)

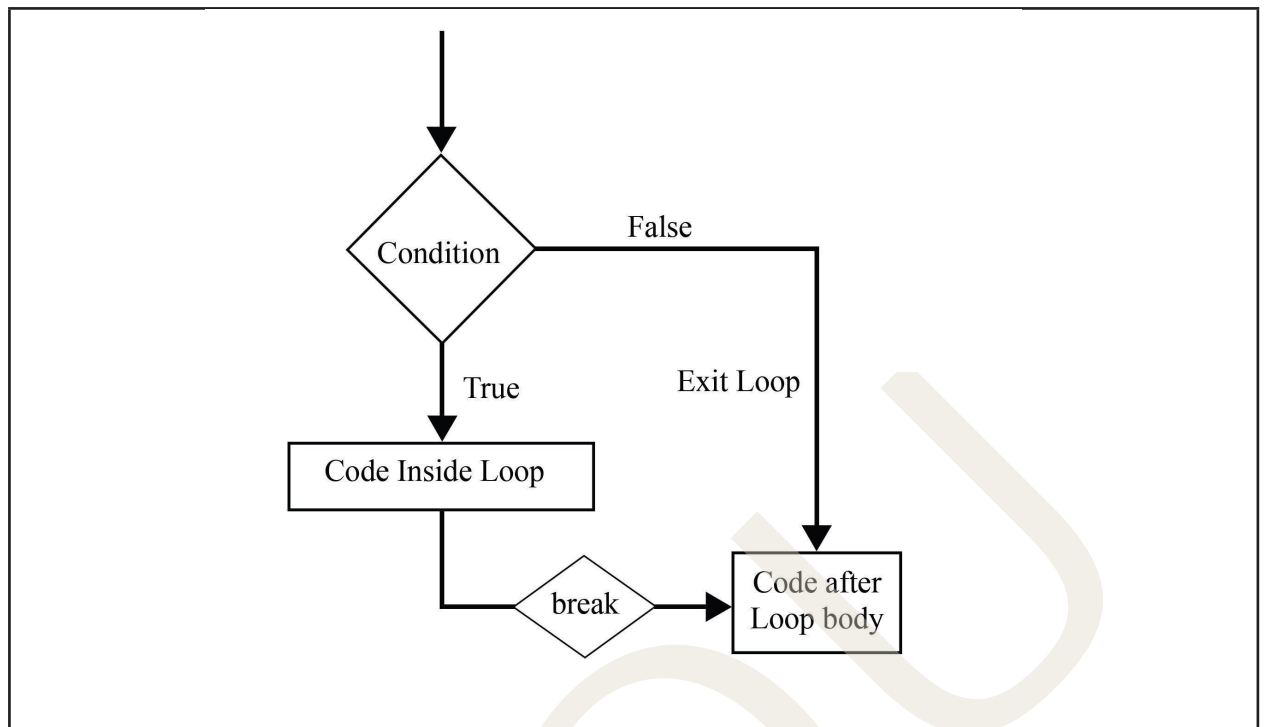


Fig 1.2.4 Flowchart for Break in Loop

Let us consider a sample program to implement the break in while loop.

```

#include <stdio.h>

int main()
{
    int num =0;
    while(num<=100)
    {
        printf("value of variable num is: %d\n", num);
        if (num >= 3)
        {
            break;
        }
        num++;
    }
    if(num <=3 )
        printf("break command executed");
}
  
```

```
    return 0;
}
```

Output:

value of variable num is: 0

value of variable num is: 1

value of variable num is: 2

break command executed

Continue statement

The continue statement is used inside loops. When it is encountered, the control skips the remaining statements in the current loop cycle and jumps directly to the beginning of the loop for the next iteration.

Syntax:

continue;

Let us consider a program

```
#include<stdio.h>

int main()
{
    int nb = 7;
    while (nb > 0)
    {
        nb--;
        if (nb == 5)
            continue;
        printf("%d ", nb);
    }
}
```

Output

6 4 3 2 1 0

In the above output you could notice that the value 5 is skipped.

goto statement

When a goto statement is used in a C program, the control transfers directly to the labeled statement mentioned in the goto.

Syntax of the goto statement is.

```
goto label_name;  
  
..  
statement(s)  
..  
  
label_name: C-statement(s);
```

The flowchart of the goto statement is shown below

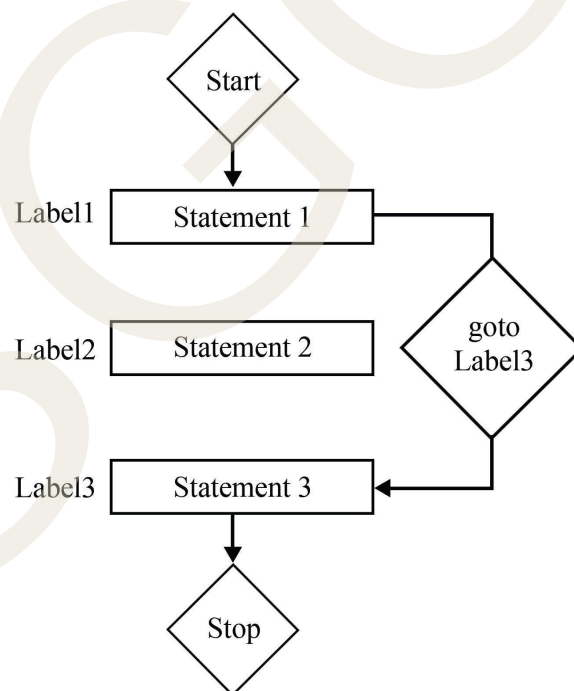


Fig. 1.2.5 Flowchart for goto Statement

Let us understand the goto statement through a c program.

```

#include <stdio.h>

#include <conio.h>

void main()

{

    int sum=0;

    for(int i = 0; i<=10; i++){

        sum = sum+i;

        if(i==5){

            goto addition;

        }

    }

    addition:

    printf("%d", sum);

    getch();

}

```

Output:

15

The goto command is an unconditional jump statement that changes the normal flow of a program. It allows the program to jump from its current position to a different part of the code. Once the jump occurs, the program continues executing from that new location and does not return to the earlier position. However, the use of goto is generally discouraged. The primary reasons for avoiding it include:

- ◆ Introduces potential bugs into the program
- ◆ Reduces readability of the code
- ◆ Makes the program logic harder to follow
- ◆ Increases difficulty during debugging
- ◆ Can usually be replaced with break and continue statements

1.2.2.5 Nested loops

A loop inside a loop is called a nested loop. Let's consider a clock which is a perfect example of a nested loop.



Fig. 1.2.6 Clock

Inside a clock, when the seconds hand completes 60 rounds, the minute hand begins its movement. After the minute hand finishes 60 rounds, the hour hand starts moving. In this context, the seconds loop is called the inner loop, while the hours loop is referred to as the outer loop.

Let us consider an example to understand the concept

```
#include <stdio.h>

int main()
{
    int i, j;
    int table = 2;
    int max = 5;
    for (i = 1; i <= table; i++)
    { // outer loop
        for (j = 0; j <= max; j++)
        { // inner loop
            printf("%d x %d = %d\n", i, j, i*j);
        }
        printf("\n"); /* blank line between tables */
    }
}
```


$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

$$1 \times 2 = 2$$

$$1 \times 3 = 3$$

$$1 \times 4 = 4$$

$$1 \times 5 = 5$$

$$2 \times 0 = 0$$

$$2 \times 1 = 2$$

$$2 \times 2 = 4$$

$$2 \times 3 = 6$$

$$2 \times 4 = 8$$

$$2 \times 5 = 10$$

The above program shows nesting of for loops. It can be extended to any level. Similarly you can do nesting of any loops.

1.2.3 Arrays

Arrays in C are one of the most fundamental and powerful tools for managing data collections. Whether you're storing a list of integers, managing a sequence of characters, or working with complex data structures, arrays provide an efficient way to handle data systematically.

Array in C is a collection of variables stored in contiguous memory locations. It can be accessed using a single name and an index. These entities or elements can be int, float, char, double, or user-defined data types, like structures.

1.2.3.1 C Array Declaration Syntax

In C, an array is a collection of elements of the same data type stored in contiguous memory locations. To declare an array, you must specify the data type, the array name, and the size (number of elements the array can hold).

The general syntax for declaring an array is:

data_type array_name[array_size];

- ◆ data_type: Specifies the type of elements the array will hold (e.g., int, float, char).
- ◆ array_name: The identifier for the array.
- ◆ array_size: The number of elements the array can hold (must be a positive integer).



1.2.3.2 Array Elements in Memory

Let's consider the following declaration: `int array[5];`

- ◆ This declaration tells the compiler to allocate space for 5 integers in memory.
- ◆ In C, the size of an int is typically 2 bytes (though it may be 4 bytes on some systems).
- ◆ So, 5 integers \times 2 bytes = 10 bytes of memory are reserved.

Since the array hasn't been initialized, the values it contains are garbage values (unpredictable data left in memory).

However, regardless of what values are stored, each element in the array is stored in a continuous block of memory. This means all elements lie next to each other in RAM.

Example:

- ◆ `int numbers[5];` // Declares an integer array named 'numbers' with 5 elements
- ◆ `float prices[10];` // Declares a float array named 'prices' with 10 elements
- ◆ `char name[20];` // Declares a character array (string) with space for 20 characters

The array size must be a constant value defined at compile time. If the size is omitted during the declaration, it must be determined from the initialization

Let us consider a C program to understand it

```
/* program to find the average of given five numbers*/
#include<stdio.h>

void main()
{
    int i, avg, num[5] = { 10,20,32,50,26},sum=0;
    for ( i = 0 ; i <5 ; i++ )
        sum = sum + num[ i ] ; /* read data from an array*/
    avg = sum / 5 ;
    printf ( "Average marks = %d\n", avg ) ;
}
```

We use the variable `i` as a subscript to refer to different elements of the array in the above code. This attribute may have several values, allowing it to apply to the array's various elements in turn. The ability to interpret subscripts with variables is what makes

arrays so useful.

Let us see how to enter elements into arrays

```
printf ( "Enter any 5 numbers " ) ;  
  
for ( i = 0 ; i < 5 ; i++ )  
{  
  
    scanf ( "%d", &num[ i ] ) ;  
  
}
```

In the above code snippet, the **for** loop repeats the process of asking the user for and obtaining any number five times. Since *i* has a value of 0 the first time around the loop, the `scanf()` function would cause the value typed to be stored in the array element `num[0]`, the array's first element. This process will be repeated before *i* attains the value of 5.

We used the “address of” operator (&) on the array element `num[i]`, just like we did on other variables, in the `scanf()` function (&*rate*, for example). We are passing the address of this particular array element to the `scanf()` function, rather than its value, as is required by `scanf()`.

The **for** loop in the following code snippet is similar, except that the body of the loop now adds each number to a running total stored in a variable called `sum`. After adding up all of the numbers, divide the total by 5, the number of students, to get the average.

```
for ( i = 0 ; i < 5 ; i++ )  
  
    sum = sum + num[ i ] ; /* read data from an array*/  
  
avg = sum / 5 ;  
  
printf ( "Average marks = %d\n", avg ) ;
```

1.2.3.3 Types of Arrays

Arrays are a fundamental data structure in C programming that stores multiple elements of the same type in a contiguous memory block. Based on their dimensions, arrays in C can be categorized into One-dimensional and Multidimensional arrays. Below, we'll explore these types in detail.

a) One Dimensional Array in C

A one-dimensional array is the simplest form of an array, storing elements in a single linear sequence. It can be visualized as a row of elements.

Declaration

```
data_type array_name[size];
```

Example: `int arr[5];` // Declares an array of size 5

Initialization

```
int arr[5] = {1, 2, 3, 4, 5};
```

Accessing Elements

```
printf("%d", arr[2]); // Accesses the third element, which is 3
```

Use Cases:

- ◆ Storing a list of numbers, such as marks or salaries.
- ◆ Simple linear data handling.

b) 2-Dimensional Array in C

A 2D array organizes data in rows and columns, making it ideal for matrix representation.

Fig 1.2.1 illustrates the representation of the following 2-D array.

e.g.: `int a[2][3] = {{1,2,3},{4,5,6}};`

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>
1	2	3
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>
4	5	6

Fig: 1.2.7 Representation of 2-D Array

Declaration

```
data_type array_name[rows][columns];
```

Example

```
int arr[3][4]; // Declares a 2D array with 3 rows and 4 columns
```

Initialization

```
int arr[2][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Points to Remember While Initializing a 2D Array:

- ◆ The total number of elements must not exceed rows * columns.
- ◆ Omitted elements are automatically initialized to 0.
- ◆ Use nested braces for better readability.

Examples of 2D Array in C: Printing a Matrix

```
#include <stdio.h>

int main() {

    int arr[2][3] = {

        {1, 2, 3},

        {4, 5, 6}

    };

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 3; j++) {

            printf("%d ", arr[i][j]);

        }

        printf("\n");

    }

    return 0;

}
```

Output

```
1 2 3
4 5 6
```

c) Multidimensional Array

Multidimensional arrays are arrays within arrays, allowing for data storage in a tabular or matrix form. They extend the concept of one-dimensional arrays by adding more dimensions. The following is an example of a multidimensional array declaration in its most basic form.

```
Datatype arrayName[size1][size2]...[sizeN];
```

The two-dimensional array is the most basic type of multidimensional array. A two-dimensional array is essentially a list of one-dimensional arrays. The following is the syntax for a two-dimensional array:

```
Datatype arrayName[size1][size2]
```

1.2.3.4 Advantages and Disadvantage of Arrays in C

a) Advantages of Arrays in C

1. **Efficient Memory Usage:** Arrays provide a way to store data sequentially, reducing memory wastage.
2. **Easy Data Access:** Elements can be accessed directly using their indices.
3. **Compact Code:** Using loops, operations can be performed on multiple elements, reducing repetitive code.
4. **Static Storage:** Data is allocated contiguously, enabling faster access and manipulation.
5. **Ideal for Fixed-Size Data:** Arrays are perfect when the number of elements is known in advance.

b) Disadvantages of Arrays in C

1. **Fixed Size:** The size of an array must be defined at the time of declaration, limiting flexibility.
2. **Homogeneous Data:** Arrays can only store elements of the same data type.
3. **No Built-in Bounds Checking:** Accessing indices outside the declared range can lead to undefined behavior.
4. **Insertion and Deletion:** These operations are time-consuming as they require shifting elements.
5. **Memory Allocation:** If improperly handled, large arrays can lead to excessive memory usage.

1.2.3.5 Strings

The C language provides a capability that enables the user to design a set of similar data types, called arrays. In the same way a group of integers can be stored in an integer array, a group of characters can be stored in a character array. Character arrays are also

at times called strings. Character arrays or strings are used by programming languages to manipulate text, such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null (`'\0'`). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies 1 byte of memory and the last character is always `'\0'`. What is this character? `'\0'` is called a null character. However `'\0'` and `'0'` are not the same. The ASCII value of `'\0'` is 0, whereas the ASCII value of `'0'` is 48. Figure 2.3.1 below shows the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations. The terminating null (`'\0'`) is important because it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a `'\0'` is not a string, but it is just a collection of characters.



Fig 1.2.8 String as an Array

However we would often use strings and hence C programming provides a shortcut for initializing strings. For example, the above example of string termed “HAESLER” could be initialized as,

```
char name[ ] = “HAESLER”;
```

In this method of declaration `'\0'` is not necessary. C automatically inserts the null character.

```
/* Program to demonstrate printing of a string */
```

```
# include <stdio.h>
```

```
int main( )
```

```
{
```

```
char name[ ] = “Hellions” ;
```

```
int i = 0 ;
```

```
while ( i <= 7 )
```

```
{
```

```
printf ( “%c”, name[ i ] ) ;
```

```
i++ ;
```

```

}

printf( "\n" );

return 0 ;

}

```

output

Hellions

Here in the above program, a character array is initialized, and then printed out the elements of this array iterating through a while loop.

1.2.4 Functions

In C programming, a function is a block of code grouped together to perform a specific task. It is enclosed within curly braces {} and can accept inputs (called parameters), process them, and return an output. Functions are extremely useful because they allow reusability, you can write a piece of code once and use it multiple times simply by calling the function with different inputs. This makes programming more efficient, organized, and modular.

Instead of writing the same logic repeatedly, you can place that logic inside a function and reuse it wherever needed. This approach helps reduce code duplication and improves readability.

Functions are important because they offer several benefits to programmers. Here are some key reasons why we use functions in C:

- ◆ They allow reusability, reducing the need to rewrite code.
- ◆ They break complex programs into simpler, manageable parts.
- ◆ They promote modularity, helping you focus on one task at a time.
- ◆ They provide abstraction, hiding internal logic from the main program.
- ◆ They make code easier to debug, maintain, and understand.

Using functions is like assembling a machine from smaller parts; each part has its role, and together they perform a larger task.

Basic Syntax of Functions

The basic syntax of functions in C programming is:

```

return_type function_name(arg1, arg2, ... argn)
{
    Body of the function //Statements to be processed
}

```


- ◆ **return_type**: Specifies the type of value the function will return (e.g., int, float). If the function doesn't return anything, use void.
- ◆ **function_name**: The name you assign to the function. It must follow naming rules in C.
- ◆ **arg1, arg2, ..., argn**: These are the parameters or inputs given to the function. A function can have no parameters or multiple parameters.

The part that includes the function name and parameter list is called the function signature.

There are two types of functions in C programming

- ◆ Built-in functions
- ◆ User-defined functions

1.2.4.1 Built-In Functions

Built-in functions are predefined functions in C. It is always available to the programmer and is also known as standard library functions.

The concept of built-in functions is similar to department sections in an office. Consider an attestation section; all persons coming for attestation purposes will be directed to this section without any uncertainty. Likewise, built-in functions have specific jobs and are grouped together in a common place called header files.

Commonly used library functions are **printf()** and **scanf()** which are included in '**stdio**' library. When using a library function in a program, we have to include the associated header files within the program.

For instance, library functions **printf()** and **scanf()** are associated with header file **stdio.h**. Before using the specified functions within the program, the header file should be linked to the program. The characteristics and definitions of built-in functions are defined within the related header file.

Syntax :

```
#include<filename.h>
```

ex: `#include<stdio.h>`

1.2.4.2 User-Defined Functions

In a C program, users can define their own functions to perform a specific task. These are code segments written by the programmer itself.

Each programming function should have input and output. The function should contain instructions or statements to process these inputs to generate the desired output. Like any other program, the input and output can be in any form such as integers, floating values, characters, arrays, etc. Before writing a user-defined function, we have to discuss some of its basic characteristics.

A user-defined function has three phases

- ◆ function declaration
- ◆ function definition
- ◆ function call

1.2.4.3 Function Declaration

Remember, before we use a variable in a program, we must declare it. Similarly, a function declaration statement is necessary before using a function.

Note that the purpose of the function declaration is to identify the function name, number and type of input parameters (*that is why parameter names are not necessary*) and return type of the function by the compiler. Function declaration statements are also known as function prototypes.

Did you notice the term parameter? Are you familiar with this? Parameters are the variables that are used during a function declaration or definition. To illustrate parameters, consider the admission procedure in a school. All the processing will be done in the office section; after that, the name list of the admitted students for a course is forwarded to the corresponding department. Here, the parameter or data is the name list prepared by the office and used by another department.

A similar concept is applicable to the concept of functions.

For example, a function to add two integers can be declared as

```
ex: int add (int x, int y);
```

The above statement declares a function with function name 'add', accepts two parameters of type integer (x and y) and returns an integer value.

It can also be written as,

```
int add(int, int);
```

Syntax: return-type function-name (list of parameters);

1.2.4.4 Function Definition

After function declaration, the code of the function has to be defined. Function definition contains the block of code to perform a specific task.

- ◆ The function definition has two parts- Function header and the Function body.
- ◆ The function header includes the function name, return type and parameter list.
- ◆ Function body is the set of instructions defined in the function definition.

Consider the function **add()** as an example. The function adds two integer numbers(a,b) and return another integer c as output.

```

int add(int a,int b)           // function name - add, input parameters - a and b,
{
    return type of the function - int
int c;

c=a+b;

return(c);                    //output parameter c
}

```

In precise, the above code segment defines a function with function name add() and the input parameters given are integer variables a and b. In the function definition, addition of parameters and the result assigned to variable c which is also an integer type.

Note: Return type of a function is the type of the variable returned by the function.

1.2.4.5 Function call

In the above two sections, we discussed function declaration and function definition. But to invoke a function to perform a specific task, the function call is needed.

Calling a function means invoking a function to do a specific task.

eg: add (5,7);

The above function call sends the values 5 and 7 to the function definition. Variables a and b (function definition) accept these values and function execution occurs.

Flow of working of a function

```

#include<stdio.h>

int add(int,int);

int add(int a, int b)
{
    int c;

    c=a+b;

    return(c );
}

void main()
{

```

```

int sum;

sum = add(5,7);

printf("Sum is %d",sum);

}

```

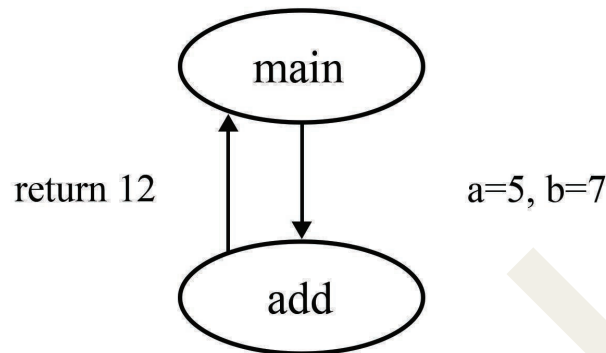


Fig 1.2.9 Working of function

How is a function implemented? What are the components? How are they interconnected? All these questions can be explained using Fig 1.2.9.

When a program execution begins, as we know, the compiler starts execution from main (). Execution proceeds till it reaches the function call statement (Step 1). When a function is called, program control is transferred to the function definition (step 2), parameters from the function call are copied to the function definition, and the code inside the function gets executed. After the execution of the return statement (Step 3) the program control is transferred to the calling function. A function can be defined either before the main() or after the main().

The following program illustrates the implementation of function

Example 1: Program to implement a function for addition of 3 integers

```

#include<stdio.h>

int add (int, int ,int);           //function declaration

int add (int a, int b,int c)       //function definition
{
    int d;

    d=a+b+c;

    return(d);

}

```

```

void main ()
{
    int w,x,y,z;

    printf ("Enter three numbers");

    scanf ("%d%d%d",&w,&x,&y);

    z=add(w,x,y);           //function call

    printf ("\nThe sum is %d",z);
}

```

Output

Enter three numbers 1 2 4

The sum is 7

The above program defines the function add() which adds three integers and returns an integer value to the calling function.

Example 2: Program to implement calculator

```

#include<stdio.h>

float add(float,float);           //function declaration

float sub(float,float);

float mul(float,float);

float div(float,float);

float add(float num_1, float num_2)    //function definition
{
    float ans;

    ans= num_1 + num_2;

    return(ans);
}

float sub(float num_1, float num_2)

```

```

{
float ans2= num_1 - num_2;
return(ans2);
}

float mul(float num_1, float num_2)
{
float ans3= num_1 * num_2;
return(ans3);
}

float div(float num_1, float num_2)
{
float ans4= num_1 / num_2;
return(ans4);
}

void main() //Program execution begins here
{
float b,c;
float a,s,m,d;
printf("Calculator\n-----\n");
printf("Enter two numbers to perform operations\n");
scanf("%f %f",&b,&c);
a=add(b,c);
printf("The result of addition is %f\n",a);
s=sub(b,c);
printf("The result of subtraction is %f\n",s);

```

```

m=mul(b,c);

printf("The result of multiplication is %f\n",m);

d=div(b,c);

printf("The result of division is %f\n",d);

}

```

The code defines an operation for addition, subtraction, multiplication and division on two variables. The functions send two float values to the function definition and execution occurs there. The results are sent back to the called functions.

Output

Calculator

Enter two numbers to perform operations

2 3

The result of addition is 5.000000

The result of subtraction is -1.000000

The result of multiplication is 6.000000

The result of division is 0.666667

Concept of functions permits to divide a program into simple and smaller tasks. It makes the code to be called many times and implements the code reusability. For example, a function to find the sum of marks can also be used by a program to find the grade of students.

Advantages of using functions

- ◆ Complexity of the program gets reduced by division of work
- ◆ Dividing a program into subprograms enables easier error handling
- ◆ A large program being divided into subprograms makes it easy to update
- ◆ Code reusability – Same code segment can be used in different programs

Nested Functions

Functions defined within other functions are called nested functions.

Syntax:

```
fun1()
```

```

{
    fun2();
}

```

The concept of nested functions can be explained through an example. Previously, we discussed the function 'add' to sum three integers (Example 1). A program that finds out the average of three numbers has to perform the same sequence of steps additionally and a division operation.

Therefore, the function to find the average can use the function 'add' to calculate the average.

```

int avg()
{
    int k, result;
    k=add();
    result=k/3;
    return(result);
}

```

Function definition of average calls the function add () and stores the sum in variable 'k'. The value of 'k' is divided by 3 and assigned as the average in variable 'result'.

Here, the function add() gets reused to find the average of numbers.

Example: Program to find the average of 5 marks

```

#include<stdio.h>
float avg();
int add();
float avg() //Nested Function
{
    int sum=add(); //function call within a function
    float avrg=sum/5;
    return(avrg);
}

```



```

int avg()
{
    int i,a[5],mark=0;

    printf("Compute average marks of 5 subjects\n-----\n");

    printf("Enter marks of 5 subjects\n");
    for(i=0;i<5;i++)
    {
        scanf("%d",&a[i]);
        mark=mark+a[i];
    }
    printf("Total mark is %d\n",mark);
    return(mark);
}

void main()
{
    float result;
    result=avg();
    printf ("The average of marks is %f", result);
}

```

Output

Compute average marks of 5 subjects

Enter marks of 5 subjects

20 30 89 90 50

Total mark is 279

The average of marks is 55.800000

Recap

- ◆ Control statements manage the flow of execution in a program
- ◆ The if statement executes a block of code when a condition is true
- ◆ If else provides two-way decision making based on a condition
- ◆ The else if ladder handles multiple conditions in sequence
- ◆ The switch statement chooses from multiple cases based on the value of a variable
- ◆ Each case in a switch must end with a colon and often a break to stop further execution
- ◆ The default case in a switch executes when no other case matches
- ◆ Loops allow repeating a block of code multiple times
- ◆ The for loop is used when the number of iterations is known
- ◆ The while loop executes as long as the condition remains true
- ◆ The do while loop executes the code block at least once before checking the condition
- ◆ The break statement exits a loop or switch early
- ◆ The continue statement skips the current iteration and starts the next one
- ◆ The goto statement jumps to a labeled line in code but should be avoided
- ◆ Arrays are collections of elements of the same type stored in contiguous memory locations
- ◆ Array elements are accessed using an index that starts from zero
- ◆ Arrays can be used to store lists of values like scores, names, or numbers
- ◆ Functions help break a large program into smaller, reusable modules
- ◆ Each function can take input (parameters), perform operations, and return output
- ◆ Using functions and loops together helps avoid code repetition and improves readability.

Objective Type Questions

1. Which statement is used to transfer program control based on a condition?
2. What keyword is used for decision making in C?
3. Which statement allows multi-way branching?
4. Which keyword is used to break out of a loop or switch block?
5. What statement skips the remaining part of the loop and proceeds to the next iteration?
6. Which loop checks the condition after executing the block at least once?
7. Which loop is best suited when the number of iterations is known?
8. What is the initial index of an array in C?
9. What is the size of an array `int arr[5]`?
10. Which bracket is used to represent an array?
11. How are arrays elements stored in memory?
12. What type of loop is a while loop?
13. Which keyword is used to declare a function with no return value?
14. What is it called when a function calls itself?
15. What is passed between functions in a function call?
16. What is the keyword used to return a value from a function?
17. What is the type of an array that stores elements of the same data type?
18. Which header file is used to handle input-output functions like `printf()`?
19. What type of control statement is if-else?
20. A one dimensional array A has indices 1....75. Each element is a string and takes up three memory words. The array is stored at location 1120 decimal. The starting address of A[49] is?
21. Array is an example of _____ type memory allocation.

22. A one dimensional array A has indices 1....75. Each element is a string and takes up three memory words. The array is stored at location 1120 decimal. The starting address of A[49] is ?
23. What will be the address of the arr[2][3] if arr is a 2-D long array of 4 rows and 5 columns and the starting address of the array is 2000?
24. What is the maximum number of dimensions an array in C can have?

Answers to Objective Type Questions

1. if()
2. if()
3. switch
4. break
5. continue
6. do..while
7. For loop
8. 0
9. 5
10. []
11. contiguous
12. entry-controlled
13. void
14. recursion
15. arguments
16. return
17. homogeneous

18. `stdio.h`
19. conditional
20. Same data type
21. Compile time
22. 1264
23. 2052
24. Theoretically there are no limits. It purely depends on system memory and compiler

Assignments

1. Explain different types of control statements in C programming with suitable examples.
2. Write a C program to display the multiplication table of a number using a loop.
3. What is an array in C? Explain how one-dimensional and two-dimensional arrays are declared and used with examples.
4. Define a function in C. How do functions enhance modular programming?
5. Differentiate between entry-controlled and exit-controlled loops with examples.
6. Write a simple program to search an element in a 1D array.
7. How to make the search easy, imagine that the array elements are sorted in order.

Reference

1. Balagurusamy, E. (2019). *Programming in ANSI C* (8th ed.). McGraw Hill Education.
2. Thareja, R. (2014). *Programming in C* (2nd ed.). Oxford University Press.

3. Kanetkar, Y. (2020). *Let Us C* (17th ed.). BPB Publications.
4. Gottfried, B. S. (2010). *Programming with C* (3rd ed.). McGraw Hill Education.

Suggested Reading

1. Perry, G., & Miller, D. (2013). *C Programming Absolute Beginner's Guide* (3rd ed.). Que Publishing.
2. Kalicharan, N. (2015). *Learn to Program with C*. Apress.
3. King, K. N. (2008). *C Programming: A Modern Approach* (2nd ed.). W. W. Norton & Company.

Unit 3

Pointers and Structures in C

Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ define what a pointer is and explain its basic purpose in C.
- ◆ describe how the dereference operator (*) is used to access the value stored at a pointer's address.
- ◆ explain how a pointer can be used to access elements of an array.
- ◆ list the basic syntax rules for declaring structure members
- ◆ familiarise common data structures that use self-referential structures.

Prerequisites

Imagine you are developing a program to manage student records in a college. Each student has several details like roll number, name, and marks, which need to be stored together. To organize this data efficiently, you use a structure that groups these related pieces of information under one name. Since there could be many students, you use pointers to store and access the memory addresses of these student records, allowing the program to quickly retrieve or update information without copying large amounts of data. This approach not only helps in handling multiple records efficiently but also lays the foundation for more complex data management techniques, such as linked lists, where each student record points to the next one, enabling dynamic and flexible data organization. Understanding pointers and structures is essential for building such practical and memory-efficient applications.

Key words

Pointer, Address, Dereference, Dot Operator, Member, Self Referential



Discussion

1.3.1 Pointers in C

A pointer is a type of variable that contains the memory address of another variable rather than the actual data. Instead of storing a value directly, it stores the location in memory where the value is kept. Pointers play a crucial role in low-level memory handling in C. When you access a pointer directly, it will display the address it holds, not the value at that address.

```
#include <stdio.h>

int main()
{
    int var = 10;           // Normal Variable
    int* ptr = &var;        // Pointer Variable ptr that stores address of var
    printf("%d", ptr);      // Directly accessing ptr will give us an address
    return 0;
}
```

Output

0x7ffa0757dd4

The memory address is represented as a hexadecimal number, beginning with 0x as in Fig 1.3.1.

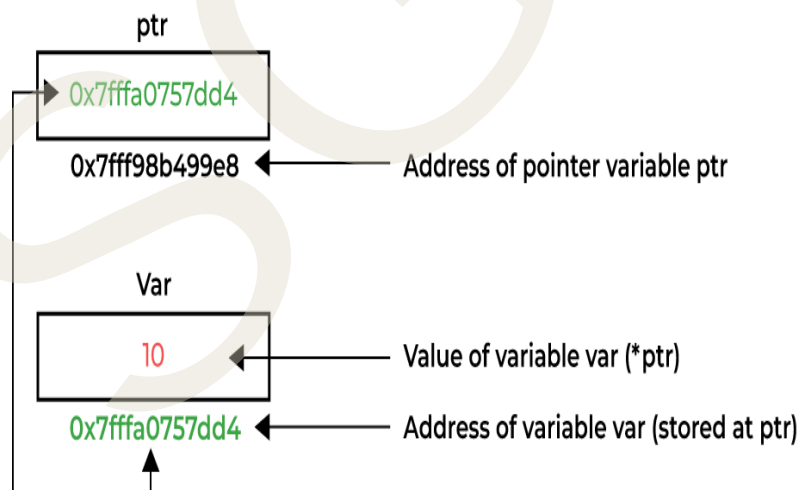


Fig. 1.3.1 Concept of pointers

1.3.2 Creation of Pointers

A pointer is defined similarly to a regular variable, but with an asterisk (*) placed before its name to indicate it is a pointer.

Syntax : data_type *pointer_name;

Here, data_type indicates the type of variable the pointer will point to. For example, an int pointer can only hold the address of an integer variable, while a float pointer can store the address of a floating-point variable.

Example : int *ptr;

This declares ptr as a pointer to an integer, meaning it can store the memory address of an integer variable. It is read as “pointer to int.”

1.3.3 Initialise the Pointers

Pointer initialization refers to the process of assigning a memory address to a pointer variable. In C, the address-of operator & is used to retrieve the memory address of a variable, which can then be assigned to a pointer. It is also possible to declare and initialize a pointer in one line, which is known as pointer definition.

Example:

```
int var = 10; // Initializing the pointer  
  
int *ptr = &var;
```

In this example, the pointer ptr holds the memory address of the variable var, obtained using the & operator.

1.3.4 Dereference Pointer

To access the value stored at the memory address held by a pointer, we need to dereference it. This is done using the dereference operator *, which is the same symbol used during pointer declaration.

```
#include <stdio.h>  
  
int main()  
{  
    int var = 10; // Store address of var variable  
    int* ptr = &var; // Dereferencing ptr to access the value  
    printf(“%d”, *ptr);  
    return 0;  
}
```

Output

10

1.3.5 Pointer to an Array

An array pointer is a pointer that refers to the entire array rather than just its first element. It treats the array as one complete unit instead of viewing it as a group of individual elements.



```
#include<stdio.h>

int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int *ptr = arr;
    printf("%p\n", ptr);
    return 0;
}
```

Output

0x7ffde273bc20

In the above example, the pointer ptr points to the first element (0th index) of the array. However, it is also possible to declare a pointer that points to the entire array rather than a single element. Such pointers are especially helpful when working with multidimensional arrays.

Syntax : type (*ptr)[size];

Where:

- ◆ type specifies the data type stored in the array.
- ◆ ptr is the name of the pointer.
- ◆ size represents the number of elements in the array the pointer refers to.

Example : int (*ptr)[10];

Here, ptr is a pointer to an array of 10 integers.

Since array subscripts ([]) have higher precedence than the dereference operator (*), the pointer name and * must be enclosed in parentheses to ensure correct interpretation.

1.3.5.1 Access Array Using Array Pointer

In the code shown below, the pointer ptr is of the type “pointer to an array of 10 integers.”

```
#include <stdio.h>

int main()
{
    int arr[3] = { 5, 10, 15 };
    int n = sizeof(arr)    //sizeof(arr[0]);
    int (*ptr)[3];        // Declare pointer variable
```

```

// Assign address of val[0] to ptr.

// We can use ptr=&val[0];(both are same)

ptr = &arr;

for (int i = 0; i < n; i++)

printf(“%d “, (*ptr)[i]);

return 0;

}

```

Output

5 10 15

1.3.6 Structures

In C, a structure is a user-defined data type that allows grouping multiple variables, possibly of different types, under a single name. It is defined using the struct keyword. The variables inside a structure are known as its members, and they can be of any valid data type. Structures are commonly used to build complex data structures like linked lists and trees. They are also helpful in modeling real-world entities in software applications, such as representing students and faculty in a college management system.

```

#include <stdio.h>           // Defining a structure

struct A
{
    int x;
};

int main()
{
    struct A a;              // Creating a structure variable
    a.x = 11;                // Initializing member
    printf(“%d”, a.x);
    return 0;
}

```

In this example, a structure named A is defined with an integer member x. A variable a of type struct A is then declared, and its member x is assigned the value 11 using the dot (.) operator. Finally, the value of a.x is displayed on the screen.

1.3.6.1 Syntax of Structure

There are two steps of creating a structure in C:

1. Structure Definition
2. Creating Structure Variables

1.3.6.2 Structure Definition

A structure is declared using the `struct` keyword, followed by the structure name and its members. This declaration acts as a structure template or prototype, and it does not allocate any memory at this stage.

```
struct structure_name  
{  
    data_type1 member1;  
    data_type2 member2;  
    ...  
}
```

`structure_name` refers to the name given to the structure.

`member1`, `member2`, ... are the identifiers used for the structure's members.

`data_type1`, `data_type2`, ... indicate the data types assigned to each member of the structure.

1.3.7 Creating Structure Variable

Once a structure is defined, a variable of that structure type must be created in order to use it. This process is similar to declaring variables of any other data type.

```
struct structure_name var;
```

We can also declare structure variables with structure definition.

```
struct structure_name {  
    ...  
} var1, var2....;
```

1.3.8 Access Structure Members

To access or modify the members of a structure, the dot operator (`.`) is used when working directly with a structure variable.

```
structure_name.member1;  
structure_name.member2;
```

However, if you are using a pointer to a structure, the arrow operator (`->`) is used to access the members.

```
structure_ptr->member1;  
structure_ptr->member2;
```

1.3.9 Self Referential Structure

Self-referential structures are structures that include a pointer as one of their members, which points to another structure of the same type. In other words, they reference their own type within their definition.

```
struct str  
{  
    int mem1;           // Reference to the same type  
    struct str* next;  
};
```

These types of structures are commonly used in various data structures, such as for defining the nodes in linked lists, trees, and similar structures.

Recap

- ◆ Pointer stores the memory address of another variable, not the actual data.
- ◆ Using %d with a pointer prints the address (in hexadecimal format).
- ◆ Syntax for pointer declaration: `data_type *pointer_name;`
- ◆ Pointers must be initialized using the address-of operator &.
- ◆ Dereferencing a pointer (*ptr) accesses the value at the memory address.
- ◆ An array pointer can point to the first element or the whole array.
- ◆ Syntax for pointer to array: `type (*ptr)[size];`
- ◆ Accessing array elements using a pointer to an array: `(*ptr)[i]`.
- ◆ A structure in C is a user-defined data type that groups variables of different types under a single name using the struct keyword.
- ◆ The variables inside a structure are called members.
- ◆ Defined using the struct keyword followed by the structure name and its members.
- ◆ Structure definition acts as a template and does not allocate memory.
- ◆ To use a structure, a structure variable must be created.

- ◆ Structure variables can be declared separately or along with the definition.
- ◆ Dot operator (.) is used to access or modify structure members through a variable.
- ◆ Arrow operator (->) is used to access structure members through a pointer.
- ◆ Self-referential structures include a pointer to a structure of the same type as one of their members.

Objective Type Questions

1. What symbol is used to declare a pointer in C?
2. Which operator is used to get the address of a variable?
3. What does a pointer store?
4. What type of pointer points to the first element of an array?
5. What is the process of accessing the value at a pointer's address called?
6. What kind of variable holds the address of another variable?
7. What type of pointer is declared as `int (*ptr)[5];`?
8. What is the term for assigning an address to a pointer?
9. What keyword is used to define a structure in C?
10. What is the term for variables inside a structure?
11. Which operator is used to access members using a structure variable?
12. Which operator is used to access members using a pointer to a structure?
13. What type of structure includes a pointer to its own type?
14. What is the default access level of structure members in C?
15. Which data structure is commonly implemented using self-referential structures?

Answers to Objective Type Questions

1. *
2. &
3. Address
4. Array
5. Dereferencing
6. Pointer
7. Array
8. Initialization
9. struct
10. Members
11. .(dot)
12. ->
13. Self-referential
14. Public
15. Linked list

Assignments

1. Explain the concept of pointers in C with a suitable example program. Describe how a pointer stores the address of a variable and how to access the value using dereferencing.
2. Write a C program to declare and initialize a pointer variable. Explain the role of the address-of operator (&) and the dereference operator (*) in your program.
3. What is a pointer to an array? Write a C program to declare a pointer to an array and access its elements using the pointer. Explain the output.

4. Write a C program to define a structure called Student with members: roll_no, name, and marks. Create an array of 5 students, accept details from the user, and display the student with the highest marks.
5. Define a self-referential structure for a singly linked list node containing an integer data field and a pointer to the next node. Write a C function to create a linked list with three nodes and display their data.

Reference

1. “The C Programming Language” by Brian W Kernighan / Dennis Ritchie
2. “Let Us C” by Yashavant Kanetkar
3. “Programming in C” by Reema Thareja
4. “Computer Basics and C Programming” by Rajaraman V

Suggested Reading

1. “Programming in ANSI C” by E. Balaguruswamy, Tata McGraw-Hill
2. C How to Program by Paul Deitel
3. Computer Science: Structured Programming Approach Using C by Behrouz A. Forouzan
4. Problem Solving and Program Design in C - With Access by Jeri R. Hanly and Elliot B. Koffman
5. C Programming Language (ANSI C) by Brian W. Kernighan and Dennis M. Ritchie

Unit 4

Dynamic memory allocation in C

Learning Outcomes

After completing this unit, the learner will be able to:

- ◆ understand the limitations of static memory allocation in C programming.
- ◆ familiarize with dynamic memory functions: `malloc()`, `calloc()`, `realloc()`, and `free()`.
- ◆ learn to allocate and manage memory during runtime.
- ◆ identify common errors and best practices in dynamic memory usage.

Prerequisites

In many real-life programming scenarios, the data we work with doesn't always come in fixed sizes. Whether you're processing user input, handling files, or working with collections of data that change during execution, flexibility becomes essential. Traditional programming techniques in C, though efficient, sometimes fall short when faced with unpredictable or growing data requirements. This is where understanding how to manage memory more effectively becomes not just useful—but necessary.

Imagine building a program that needs to store student marks, inventory items, or sensor readings, but you don't know beforehand how many entries there will be. What happens when the amount of data exceeds the capacity you initially planned for? Or when you've reserved more memory than you actually need? These are the types of challenges that prepare the ground for an important and powerful feature in C programming - one that gives you greater control over memory and helps you build smarter, more efficient code.

Key words

Static memory allocation, Dynamic memory allocation, `malloc()`, `calloc()`, `free()`, `realloc()`



Discussion

1.4.1 Introduction

C is a structured programming language that allows fine control over system resources such as memory. One limitation of conventional memory management in C is the fixed size of arrays. Arrays in C are declared with a specific size during compilation, and this size cannot be changed during program execution. This restriction often leads to inefficient memory use either wasting memory when arrays are over-provisioned or causing program crashes or data loss when arrays are under-provisioned.

Dynamic memory allocation is an essential feature in C that allows programs to manage memory more efficiently during execution. Unlike statically allocated arrays, which have a fixed size determined at compile time, dynamic memory allocation enables the program to request and release memory as needed at runtime. This flexibility helps avoid problems like wasted memory due to over-allocation or insufficient memory when the data size exceeds initial estimates. Using dynamic memory functions, C programmers can build more adaptable and memory-efficient applications.

1.4.2 Why Dynamic Memory Allocation is Needed

Consider an integer array defined as:

```
int arr[9];
```

Here, memory is allocated at compile time for 9 integers, typically totaling 36 bytes (assuming each integer takes 4 bytes). This is called Static memory allocation. However, in many real-life scenarios, the exact amount of data is not known in advance. This leads to two common problems:

Case 1: Memory Waste

If the user inputs only 5 values, then 4 integer slots go unused, wasting memory.

Case 2: Insufficient Size

If the user needs to input 12 values, but the array size is fixed at 9, the program cannot accommodate the extra values without rewriting or reallocating the array.

To solve both these problems, **Dynamic Memory Allocation** comes into play. It allows the program to allocate or free memory during runtime, depending on the current requirements.

1.4.3 What is Dynamic Memory Allocation ?

Dynamic Memory Allocation is the process of allocating memory storage during runtime, as opposed to compile-time. C provides a set of standard library functions (defined in <stdlib.h>) to facilitate dynamic memory allocation

1. malloc() – Memory Allocation
2. calloc() – Contiguous Allocation
3. free() – Deallocation

4. realloc() – Reallocation

1.4.3.1 malloc() – Memory Allocation

malloc() is used to allocate a block of memory of a given size (in bytes) from the heap. It returns a void pointer, which can be cast to the desired type.

Syntax

```
ptr = (cast-type*) malloc(byte-size);
```

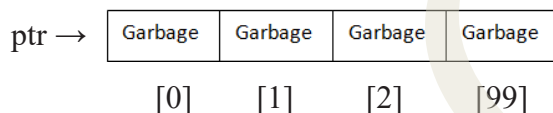
Example

```
int* ptr;
```

```
ptr = (int*) malloc(100 * sizeof(int));
```

Here, malloc allocates space for 100 integers (400 bytes if int = 4 bytes). ptr now holds the address of the first byte in the block.

One important point to remember is that malloc() does not initialize the allocated memory block. As a result, the contents of the allocated memory contain garbage values. If the allocation fails, malloc() returns a NULL pointer.



1.4.3.2 calloc() – Contiguous Allocation

calloc() is used to allocate multiple contiguous blocks of memory and initializes all bits to zero.

Syntax

```
ptr = (cast-type*) calloc(num_elements, element_size);
```

Example

```
float* ptr;
```

```
ptr = (float*) calloc(25, sizeof(float));
```

This allocates space for 25 floats, each initialized to zero.

Comparison with malloc()

Feature	malloc()	calloc()
Initialization	Not initialized	Initialized to zero
Parameters	Single	Two (elements, size per element)
Memory Layout	Contiguous	Contiguous

ptr →	0.0	0.0	0.0	0.0
	[0]	[1]	[2]	[24]

1.4.3.3 free() – Freeing allocated memory

Dynamically allocated memory must be manually deallocated. Otherwise, it leads to memory leaks. free() releases the previously allocated memory back to the heap.

```
free(ptr);
```

Example

```
int* ptr = (int*) malloc(100 * sizeof(int));
```

```
// ... use the memory
```

```
free(ptr); // deallocates memory
```

It is good practice to set the pointer to NULL after freeing. Double freeing (calling free() twice on the same pointer) should be avoided.

1.4.3.4 realloc() - Reallocation

realloc() is used to resize a memory block previously allocated with malloc() or calloc().

Syntax

```
ptr = realloc(ptr, newSize);
```

Example

```
int* ptr = (int*) malloc(5 * sizeof(int));
```

```
// Later, need space for 10 elements
```

```
ptr = (int*) realloc(ptr, 10 * sizeof(int));
```

If the new size is larger, realloc() expands the block and leaves original data unchanged (new space contains garbage). If the new size is smaller, it truncates the block. If memory cannot be extended, it may move the entire block to a new location.

Before:

ptr →	10	20	30	40	50
-------	----	----	----	----	----

After realloc to 8:

ptr →	10	20	30	40	50	Garbage	Garbage	Garbage
-------	----	----	----	----	----	---------	---------	---------

1.4.4 Common errors in dynamic memory allocation

Error Type	Description
Memory Leak	Forgetting to use free()
Dangling Pointer	Accessing memory after free()
Uninitialized Access	Using memory without initializing
Double Free	Calling free() twice on the same pointer

1.4.5 Best Practices in Dynamic Memory Allocation

Always check for NULL:

```
if (ptr == NULL) {  
    printf("Memory allocation failed.\n");  
    exit(1);  
}
```

Free memory after use:

```
free(ptr);  
  
ptr = NULL;
```

Avoid memory leaks by ensuring every malloc() or calloc() is eventually paired with a free().

Avoid dereferencing freed memory:

```
free(ptr);  
  
// Do not use ptr now unless reassigned.
```

1.4.6 Static vs Dynamic memory allocation

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Time of Allocation	At compile time	At runtime
Memory Management	Done automatically by the compiler	Done manually by the programmer using functions
Flexibility	Fixed size (cannot be changed after allocation)	Flexible (can increase or decrease memory as needed)
Location (Storage Area)	Stack (for local variables), or data segment (for global/static)	Heap (dynamic memory area)

Function Used	No special function needed	Uses malloc(), calloc(), realloc(), and free()
Initialization	Automatically initialized (for global/static); not for local	malloc() – garbage values, calloc() – initialized to zero
Memory Efficiency	May lead to wasted or insufficient memory	More memory-efficient, as it allocates exactly what's needed
Deallocation	Not required (done automatically)	Must be done manually using free()
Error Handling	Fewer memory errors	Risk of memory leaks, dangling pointers if not handled properly
Example	int arr[10];	int* arr = (int*) malloc(10 * sizeof(int));

Illustrative Program using all Dynamic Memory allocation Functions

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int* ptr;

    int n, i;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    // Allocating memory using malloc

    ptr = (int*) malloc(n * sizeof(int));

    if (ptr == NULL) {

        printf("Memory not allocated.\n");

        exit(0);

    }

    // Initializing elements

    for (i = 0; i < n; ++i) {

        ptr[i] = i + 1;
```

```

    }

    printf("Array elements:\n");

    for (i = 0; i < n; ++i) {

        printf("%d ", ptr[i]);

    }

    // Reallocating memory

    printf("\nEnter new size: ");

    scanf("%d", &n);

    ptr = realloc(ptr, n * sizeof(int));

    for (i = 0; i < n; ++i) {

        ptr[i] = i + 10;

    }

    printf("Modified elements:\n");

    for (i = 0; i < n; ++i) {

        printf("%d ", ptr[i]);

    }

    free(ptr); // deallocating memory

    return 0;
}

```

Recap

- ◆ Fixed-size arrays cannot grow or shrink during program execution.
- ◆ Leads to two problems:
- ◆ **Wastage of memory** if array size is larger than needed.
- ◆ **Insufficient memory** if more elements are needed than allocated.
- ◆ Real-world applications often require handling variable-sized data.
- ◆ Static memory allocation is not flexible for such scenarios.
- ◆ **Dynamic Memory Allocation** solves this by allocating memory **at runtime**.
- ◆ Allows memory to be **allocated**, **resized**, and **freed** as needed.
- ◆ Provides efficient memory usage and better adaptability in programs.
- ◆ Four key functions (from `<stdlib.h>`):
 - `malloc()` – allocates memory block.
 - `calloc()` – allocates and initializes memory.
 - `realloc()` – resizes previously allocated memory.
 - `free()` – deallocates memory.
- ◆ Returns a pointer to the allocated block.
- ◆ Must be manually managed to avoid memory issues.

Objective Type Questions

1. Which memory allocation function initializes memory to zero?
2. Which header file contains dynamic memory allocation functions?
3. What does `malloc()` return if memory allocation fails?
4. In which memory segment does dynamic memory allocation occur?
5. What type of pointer does `malloc()` return?
6. Which function is used to deallocate dynamically allocated memory?
7. What is the default value of memory allocated by `malloc()`?

8. Which function resizes previously allocated memory?
9. What kind of error occurs if memory is not freed?
10. Which function can be used to allocate memory for multiple blocks?

Answers to Objective Type Questions

1. calloc
2. stdlib.h
3. NULL
4. Heap
5. void
6. free
7. Garbage
8. realloc
9. Memory leak
10. calloc

Assignments

1. Compare and contrast static and dynamic memory allocation in C.
2. Explain the concept of dynamic memory allocation in C.
3. Write a C program that demonstrates the use of all four dynamic memory functions (malloc(), calloc(), realloc(), and free()).

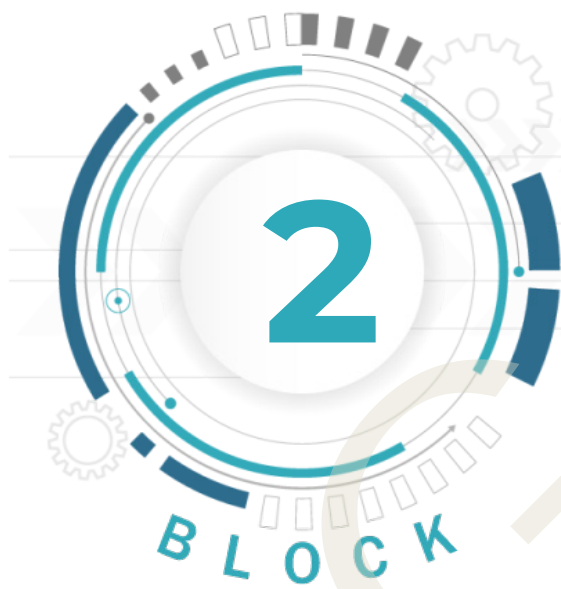
Reference

1. Kanetkar, Y. (2008). *Let Us C* (10th ed.). BPB Publications.
2. Prata, S. (2014). *C Primer Plus* (6th ed.). Addison-Wesley.
3. King, K. N. (2008). *C Programming: A Modern Approach* (2nd ed.). W. W. Norton & Company.

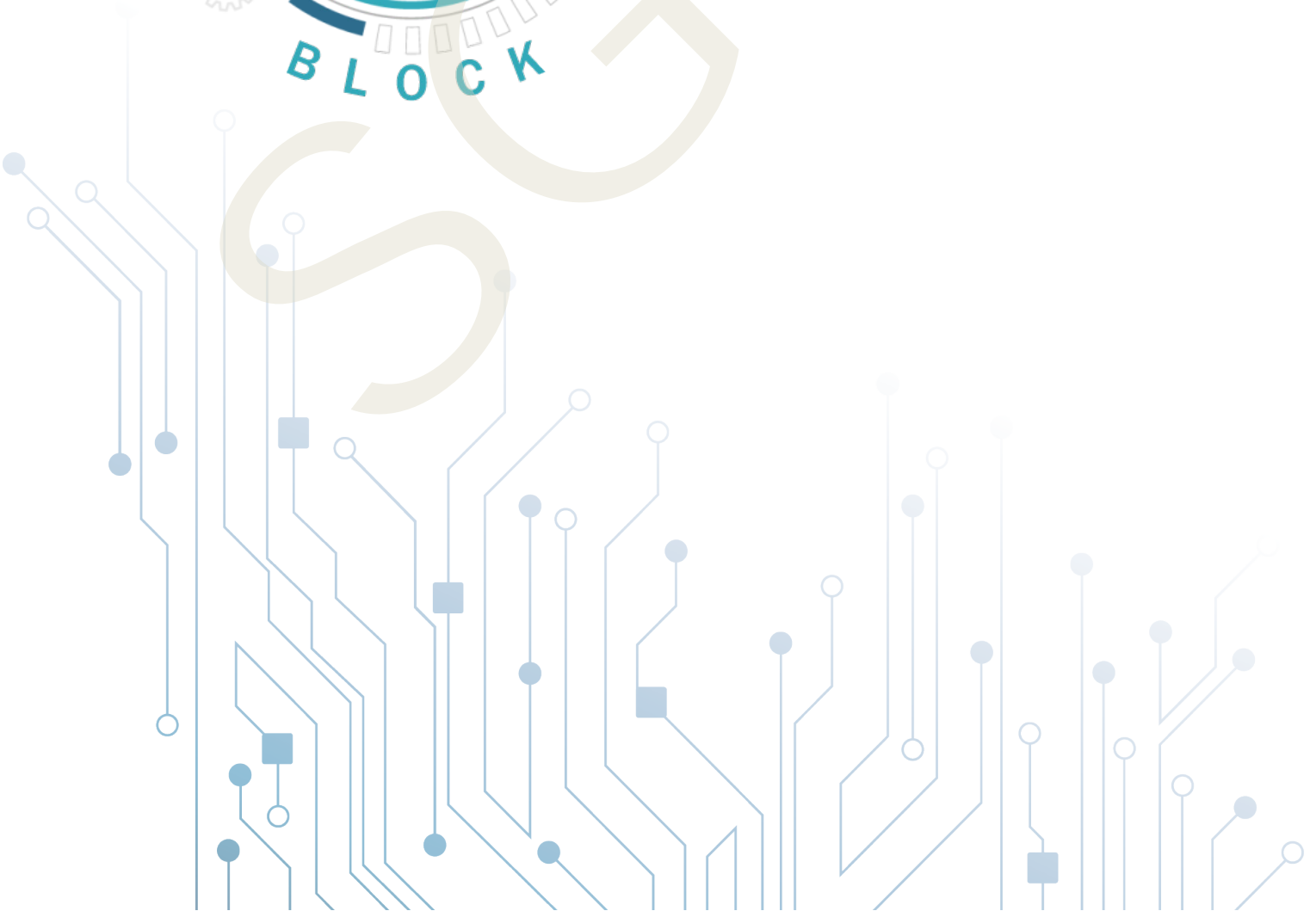
4. Malik, D. S. (2009). *C Programming: From Problem Analysis to Program Design* (5th ed.). Cengage Learning.
5. <https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>
6. https://www.tutorialspoint.com/cprogramming/c_dynamic_memory_allocation.htm

Suggested Reading

1. C programming: Kernighan and Ritchie
2. Programming with C - B.S. Gottfried, Schaum's Outline Series, Tata McGraw-Hill, 2006



Basic Data Structures



Unit 1

Introduction to Data Structures

Learning Outcomes

Upon completion of this unit, the learner will be able to:

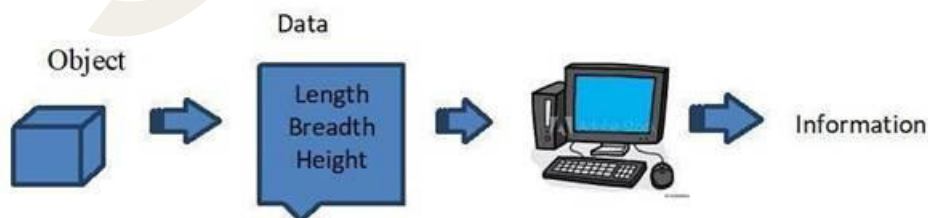
- ◆ identify the need of various data types and data structures
- ◆ explain the classification of data structures.
- ◆ familiarize the concept of linear and nonlinear data structures.
- ◆ familiarize static and dynamic memory allocation

Prerequisites

We are living in a world of data and information. In every second, every minute, a huge amount of data is emerging from different sectors like business, agriculture, industry, information technology and social media. The study of computer application encompasses the study of organization, flow and processing of the data in various sectors using a computer. The data structure is a tool used to process digital data in an efficient manner, efficiency in terms of time and space.

Data and Information

Do you know what the relationship between data and information is? In computing data is a sequence of symbols that represents an object (example student, vehicle etc.), a relationship, or an idea. Data represented using binary numbers zero and one is called a digital data. Processed data is called information. So processing digital data we get digital information. Modern computers process digital data to get digital information.



The figure illustrates the relationship between data and information with an example. Here we can see a cube, an object and data derived from the cube length, breadth and

height. From this data we can compute the volume, the surface area etc. the different information regarding the data.

Data types

Data type is a concept that defines internal representation of data in memory. Every programming language has its own set of data types. The basic data type of programming language is called primitive data type. Integer, character and float/Real are examples for primitive data types. Data types formed using primitive data types are called derived data types. Pointers, structure and union are examples for derived data types.

Abstract data types

In computer science, an abstract data type (ADT) is a mathematical model for data types. It is defined by a set of possible values and operations on a data. In the example specified above the set $\langle \text{length, breadth, height} \rangle$ is an ADT and from this particular ADT we can compute the volume of the cube where volume is the product of three parameters length, breadth and height.

Key words

Queue, Tree, Stack, Array, Linkedlist, Graph

Discussion

We learned that an algorithm is a sequence of steps that a program or any computational procedure has to take. A program on the other hand is an implementation of an algorithm and it could be in any programming language. During execution, programs take different inputs or data. Data structure is the way we need to organize the data, so that it can be used effectively by the program. Thus we can define data structure as an Organization of data that is needed to solve the problem. Array, Stacks, Queue, List and Graphs are examples for data structure that are widely used for design and analysis of step by step processing of data.

Need of data structure

As applications are getting more complex and amount of data is increasing by time, there may arise the following problems:

Processor speed: To handle a very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processors may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store; if our application needs to search for a particular item, it needs to traverse 106 items every time, resulting in slowing down the search process.

Multiple requests: If thousands of users are simultaneously searching data on a web

server, there is a risk that even a very large server may fail during this process.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

2.1.1 Classification of Data Structure

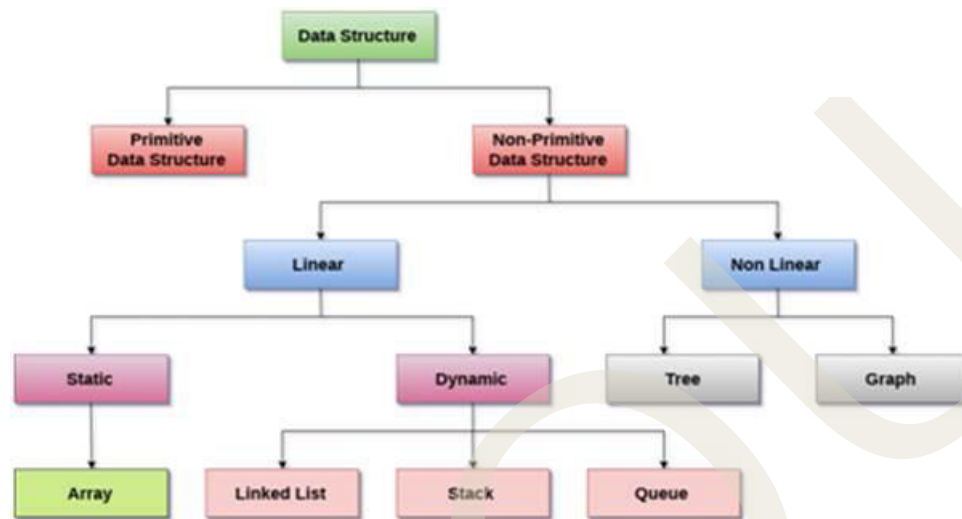


Fig 2.1.1 Classification of data structure

Data Structures are generally classified into two classes:

- ◆ Primitive Data Structures
- ◆ Non-primitive Data Structures

2.1.2 Primitive Data Structures

Primitive Data Structures are the fundamental data types which are supported by programming languages. They are created without the support of other data structures as a support or tool.

A primitive data structure is a basic type of data structure that stores data of a single type.

The primitive data structure consists of fundamental data types like float, character, integer, etc.

Examples:

Integer	: 1, 2, 3, -1, -4, ...
Real(float)	: 2, 0.234, $\sqrt{3}$, ...
Characters	: a, b, A, B, C, ...
Booleans	: 0 and 1

2.1.3 Non-Primitive Data Structures

Non-primitive Data Structures are created using primitive data structures. Non-primitive data structures are more complicated data structures and are derived from primitive data structures. These Data Structures can be designed by users. Examples: Lists, Graphs, Stacks and Trees.

A non-primitive data structure is considered as the user-defined structure that allows storing values of different data types within one entity.

Non-Primitive Data Structures can further be classified into two categories:

- ◆ Linear Data Structures and
- ◆ Non-Linear Data Structures

and is discussed below.

2.1.4 Linear Data Structure

A data structure is called linear if all of its elements are arranged in the sequential order. In linear data structures, the elements are stored in a non-hierarchical way where each element has the successors and predecessors except the first and last element. It is further divided into two:

- ◆ Static
- ◆ Dynamic

2.1.5 Static Data Structure

A static data structure is an organization or collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later point. Arrays are a prominent example of a static data structure.

◆ Array

An array is a collection of similar types of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

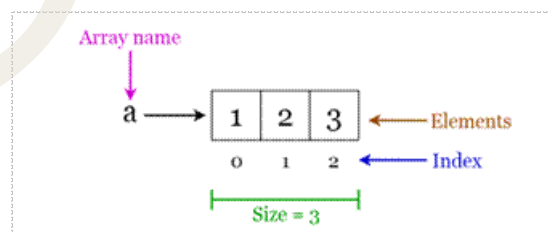


Fig 2.1.2 Visualization of basic terminology of array

2.1.6 Dynamic Data Structure

In Dynamic data structure, the size of the structure is not fixed and can be modified

during its operations. Dynamic data structures are designed to facilitate change of data structures during run time.

Examples:

◆ Linked List

Linked list is a linear data structure which is used to maintain a list in the memory. It can be viewed as a collection of nodes stored at different memory locations that are not contiguous. Each node of the list contains a pointer to its adjacent node as shown in the figure 1.1.7.

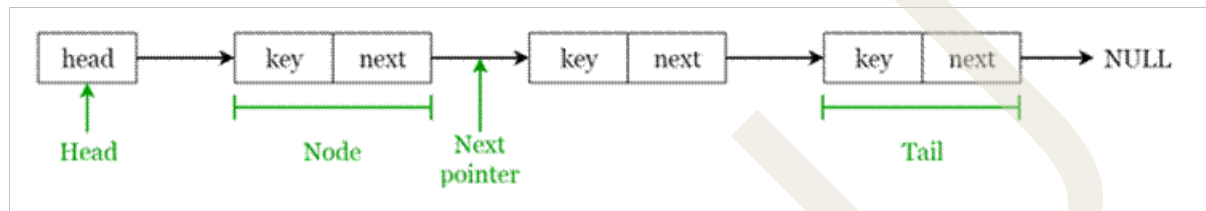


Fig 2.1.3 Visualization of basic terminology of linked list

◆ Stack

We can visualize a stack like a pile of plates placed one on top of the other. Each plate below the topmost plate cannot be directly accessed until the plates above are removed. Plates can be added and removed from the top only. Each plate is an element and the pile is the stack. In the programming terms, each plate is a variable and the pile is a data structure.

A stack is a linear data structure which follows Last-In-First-Out (LIFO) principle



Fig 2.1.4 Visualization of stack as a pile of plates

Stack is a linear data structure in which insertion and deletions are allowed only at one

end, called top. It follows Last-In-First-Out (LIFO) methodology for storing the data items. PUSH() and POP() are the two important operations related to stack. PUSH() operation is used to insert new data items into the stack and POP() operation is used to remove or delete a data item from the stack.

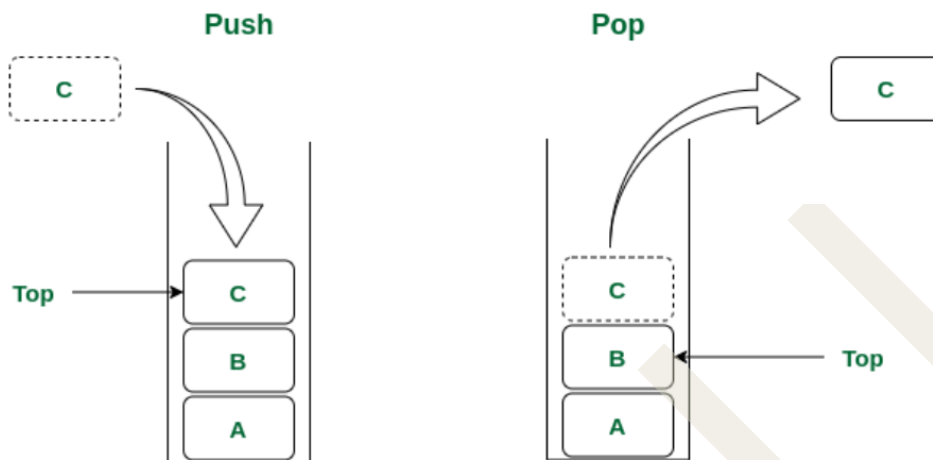


Fig 2.1.5 Visualization of stack data structure

◆ Queue

Queue is a linear data structure in which elements can be inserted only at one end called rear and deleted only at the other end called front. It is an abstract data structure, similar to stack. Queue is opened at both ends therefore it follows First-In-First-Out (FIFO) methodology for storing the data items. Data insertion operation in a queue is known as enqueue and data deletion operation in a queue is known as dequeue. It is like the passengers standing in a queue to board a bus. The person who first gets into the queue is the one who first gets on the bus. The new passengers can join the queue from the back whereas passengers get on the bus from the front.

A queue is a linear data structure which follows First-In-First-Out (FIFO) principle

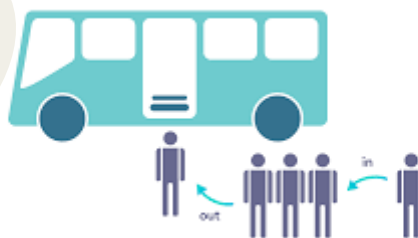


Fig 2.1.6 Visualization of Queue

2.1.7 Non Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not

arranged in sequential structure. Examples of Non Linear Data Structures are given below:

◆ Trees

Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf nodes while the topmost node is called root node. Each node contains pointers to point adjacent nodes. Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node as shown in figure 2.1.7.

The files and folders in Windows Explorer are organized in a tree format

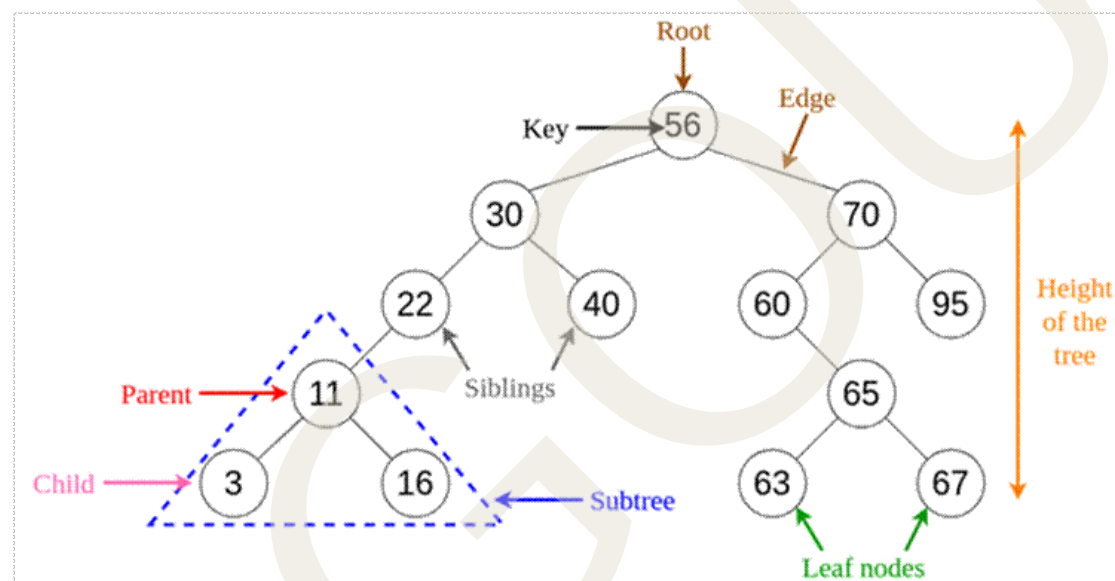


Fig 2.1.7 Visualization of basic terminology of trees

◆ Graphs

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from a tree in the sense that a graph can have cycles, while a tree cannot have cycles.

A Graph is a network of interconnected items. Each item is known as a node and the connection between them is known as the edge. You probably use social media like Facebook, LinkedIn, Instagram, and so on. Social media is a great example of a graph being used. Social media uses graphs to store information about each user. Here, every user is a node just like in Graph. And, if one user, let's call him Jack, becomes friends with another user, Rose, then there exists an edge (connection) between Jack and Rose. Likewise, the more we are connected with people, the nodes and edges of the graph keep on increasing.

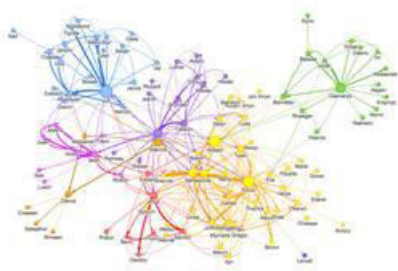


Fig 2.1.8 Visualization of graph

Similarly, Google Map is another example where Graphs are used. In the case of the Google Map, every location is considered as nodes, and roads between locations are considered as edges. And, when one has to move from one location to another, the Google Map uses various Graph-based algorithms to find the shortest path.

Graphs may be directed or undirected as shown in figure 2.1.9(a) and figure 2.1.9(b). A graph is said to be a directed graph if all its edges have a direction indicating the start vertex and the end vertex. Observe figure 2.1.9(a), it is a directed graph with edge set E and vertex set G . We say that the edge $(1, 2)$ is incident from or leaves vertex 1 and is incident to or enters vertex 2 as shown in the figure 2.1.9(a). Edges from a vertex to itself are called Self-loops. In the figure 2.1.9 (a) in vertex 2 there is a self-loop.

A graph is defined as a pair $G = (V, E)$, where V is a set of elements called vertices, and E is a set of pairs of vertices, called edges.

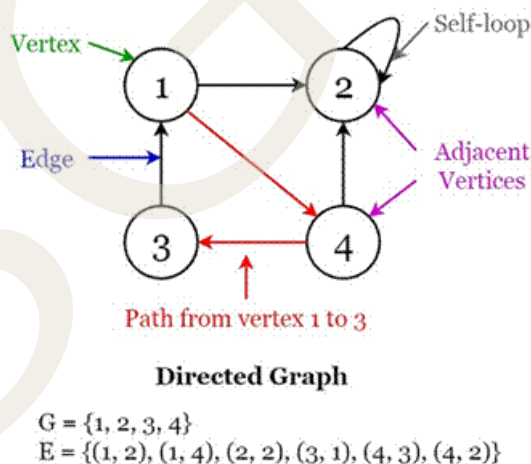


Fig 2.1.9(a) Visualization of basic terminology of Directed graph

A graph is said to be an undirected graph if all its edges have no direction as shown in figure 2.1.9(b). It can traverse in both directions between the two vertices. If a vertex is not connected to any other node in the graph, it is called an isolated vertex as shown in the figure 2.1.9(b).

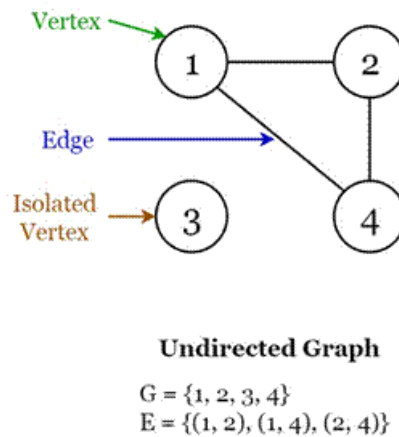


Fig 2.1.9(b) Visualization of basic terminology of undirected graph

2.1.8 Contiguous and Non-Contiguous Memory Allocation

Memory allocation refers to the process of reserving space in the computer's memory to store data or program instructions. In data structures, memory allocation is crucial as it affects the performance and behavior of data structures. There are two major types of memory allocation techniques: Contiguous Memory Allocation and Non-Contiguous Memory Allocation.

2.1.8.1 Contiguous Memory Allocation

Contiguous memory allocation is a method where all the data elements of a data structure are stored in a single continuous block of memory. This means that the memory locations are next to each other, without any gaps.

Examples:

Arrays and matrices are common data structures that use contiguous memory allocation. Each element can be accessed directly using its index.

Advantages:

- ◆ Fast access using index values.
- ◆ Simple to implement.
- ◆ Good performance due to locality of reference.

Disadvantages:

- ◆ Requires knowing the size in advance.
- ◆ Difficult to insert or delete elements.
- ◆ Can lead to memory wastage if allocated space is unused.

Example Explanation:

Consider an array of 5 integers: [10, 20, 30, 40, 50]. If the base address of the array is 1000 and each integer takes 4 bytes, the elements will be stored at consecutive

addresses: 1000, 1004, 1008, 1012, and 1016.

2.1.8.2 Non-Contiguous Memory Allocation

Non-contiguous memory allocation is a method where elements are stored in different memory locations and are connected using pointers or references. This allows memory to be used more flexibly and efficiently, especially when the size of the data is not known in advance.

Examples:

Linked lists, trees, and graphs are data structures that use non-contiguous memory allocation. Each node contains data and a pointer to the next node.

Advantages:

- ◆ Dynamic size adjustment.
- ◆ Efficient insertion and deletion.
- ◆ Better use of available memory.

Disadvantages:

- ◆ Slower access time compared to arrays
- ◆ Requires extra memory for storing pointers.
- ◆ More complex to implement and manage.

Example Explanation:

In a linked list, each element (node) contains data and a pointer to the next node. These nodes may be scattered in memory, but the pointers help maintain the correct order.

2.1.9 Static and Dynamic memory allocation

Memory allocation refers to the process of assigning memory to variables or data structures so they can be used by programs during execution. Based on when and how memory is allocated, it is classified into two types: Static Memory Allocation and Dynamic Memory Allocation. These approaches determine whether the memory size is fixed at compile-time or flexible at run-time.

2.1.9.1 Static Memory Allocation

Static memory allocation means that memory for variables or data structures is allocated before program execution, usually at compile time. Once the memory is allocated, the size remains fixed and cannot change during the execution of the program.

Examples

Arrays declared with a fixed size such as `int arr[100]`; in C or C++ use static memory allocation. The size of 100 is fixed and cannot be altered during the program's run.

Advantages

- ◆ Memory is allocated at compile time, so no overhead during execution.



- ♦ Faster access because the memory layout is known in advance.
- ♦ Easy to implement and manage.

Disadvantages

- ♦ Memory size must be known in advance
- ♦ Inefficient if the allocated memory is not fully used.
- ♦ Inflexible – memory cannot be resized or adjusted while the program is running.

2.1.9.2 Dynamic Memory Allocation

Dynamic memory allocation allows memory to be assigned during program execution, i.e., at run-time, based on the current needs of the program. This makes it possible to create flexible and resizable data structures.

Examples

In C, functions like `malloc()`, `calloc()`, and `realloc()` are used for dynamic memory allocation. In C++, the `new` keyword is used. In Python, lists are implemented dynamically.

Example in C:

```
int* ptr = (int*)malloc(10 * sizeof(int));
```

This allocates memory for 10 integers during program execution.

Advantages

- ♦ Flexible memory usage – memory can be increased or decreased as needed.
- ♦ Suitable for programs where memory requirements vary at runtime.
- ♦ Efficient use of memory when the exact size is not known in advance.

Disadvantages

- ♦ Slightly slower performance due to allocation overhead at run-time.
- ♦ Requires careful management (e.g., freeing memory) to avoid memory leaks.
- ♦ More complex to implement and debug.

Table 2.1.1 Comparison of static and dynamic

Feature	Static Memory Allocation	Dynamic Memory Allocation
Allocation Time	Compile-time	Run-time
Memory Size	Fixed	Flexible

Example	Fixed-size array	Linked list, dynamic array
Performance	Fast	Slightly slower
Memory Efficiency	May cause wastage	Efficient if used carefully
Flexibility	Low	High

Recap

- ◆ Data structures help organize and manage data efficiently for processing and storage.
- ◆ Data is raw input; information is processed, meaningful data.
- ◆ Primitive data types are basic types like int, float, char, and bool.
- ◆ Non-primitive data types include linear (arrays, stacks, queues, linked lists) and non-linear (trees, graphs) structures.
- ◆ Linear data structures store elements sequentially, either statically or dynamically.
- ◆ Static data structures have a fixed size allocated at compile time.
- ◆ Dynamic data structures can grow or shrink during runtime.
- ◆ Arrays use contiguous memory allocation for fast access but fixed size.
- ◆ Linked lists, trees, and graphs use non-contiguous memory with pointers for flexibility.
- ◆ Static memory allocation is done before execution with fixed memory size.
- ◆ Dynamic memory allocation happens during execution and adjusts to memory needs.

Objective Type Questions

1. Which data structure follows the LIFO principle?
2. Which data structure follows the FIFO principle?
3. What type of data structure is an array?
4. Which memory allocation type stores elements in adjacent memory blocks?
5. Which memory allocation type allows dynamic resizing?
6. What is the basic data type used to build other complex types?
7. Which data structure organizes data in a hierarchical manner?
8. What is a basic unit of a graph called?
9. What is another term for dynamic memory allocation during runtime?
10. Which graph allows edges to have directions?

Answers to Objective Type Questions

1. Stack
2. Queue
3. Linear
4. Contiguous
5. Non-contiguous
6. Primitive
7. Tree
8. Vertex
9. Heap
10. Directed

Assignments

1. Define a data structure. Why is it important in computer programming?
2. Differentiate between primitive and non-primitive data structures with examples.
3. Explain the relationship between data and information with a suitable example.
4. Describe the characteristics and examples of linear and non-linear data structures.
5. What is the difference between static and dynamic memory allocation? Explain with examples.
6. Write a short note on contiguous and non-contiguous memory allocation. Give advantages to each.
7. Explain the working of stack and queue data structures with real-life examples.
8. What is a linked list? How is it more efficient than arrays in terms of memory usage?

Reference

1. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). *Fundamentals of data structures in C* (2nd ed.). Universities Press.
2. Weiss, M. A. (2007). *Data structures and algorithm analysis in C* (2nd ed.). Pearson Education.
3. Thareja, R. (2014). *Data structures using C* (2nd ed.). Oxford University Press.
4. Srivastava, S. K., & Srivastava, D. (2009). *Data structures through C in depth*. BPB Publications.
5. Kanetkar, Y. (2020). *Let us C* (17th ed.). BPB Publications.

Suggested Reading

1. GeeksforGeeks. (n.d.). *Data structures*. <https://www.geeksforgeeks.org/data-structures>
2. Programiz. (n.d.). *Data structures tutorial*. <https://www.programiz.com/dsa>
3. TutorialsPoint. (n.d.). *Data structures and algorithms tutorial*. https://www.tutorialspoint.com/data_structures_algorithms
4. National Programme on Technology Enhanced Learning (NPTEL). (n.d.). *Data structures and algorithms*. <https://nptel.ac.in>

SGOU

Unit 2

Array as a Data Structure

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- ♦ define an array and identify its key attributes such as array name, array size, and array type.
- ♦ list the types of arrays, including one-dimensional, two-dimensional, and multi-dimensional arrays.
- ♦ recall the syntax used to declare and initialize one-dimensional and two-dimensional arrays in C.
- ♦ explain how one-dimensional and two-dimensional arrays are stored in memory.
- ♦ identify the advantages and disadvantages of using arrays as a data structure.

Prerequisites

Before you begin learning about arrays, it's important to recall some basic programming concepts that you are already familiar with.

You already know how to declare variables to store individual values in C. For example:

```
int age = 20;  
float price = 99.5;  
char grade = 'A';
```

Each of these variables holds a single value. But in real-world applications, we often need to store and manage multiple values of the same type such as the marks of 50 students, temperatures recorded every hour, or the names of several books. Creating a separate variable for each value would be inefficient and difficult to manage.

You are also familiar with loops, such as for and while, which allow you to repeat actions. These loops become even more useful when working with large sets of data.

To solve the problem of storing many related values efficiently, programming languages like C provide a data structure called an array. An array allows you to store multiple



values of the same type under a single variable name, with each value accessed using an index.

Arrays make it easier to manage data in an organized way. They support powerful operations like searching, sorting, updating, and traversing using loops. Arrays are also useful in solving real-world problems such as processing lists, handling tabular data, and performing mathematical calculations using matrices.

Key words

Arrays, One-dimensional array, Two-dimensional array, Memory representation, Array declaration

Discussion

2.2.1 Array as a data structure

In data structures, arrays are commonly used to store data of the same type under a single name. They are particularly effective for handling lists and tables.

As shown in Figure 2.2.1, an array containing English alphabets is stored in the computer's memory. The first element in the array is 'U', which is at index 0 and stored at memory location 200. Each element in the array has a corresponding index. These elements are accessed using their specific index values.

An array with n elements will have indices ranging from 0 to $n-1$. The smallest index is known as the lower bound, and the largest index is the upper bound. It's important to note that all elements in an array are stored in adjacent memory locations. Therefore, arrays are said to be contiguous in memory.

An array is a linear data structure that gathers elements of the same data type, storing them in contiguous and adjacent memory locations.

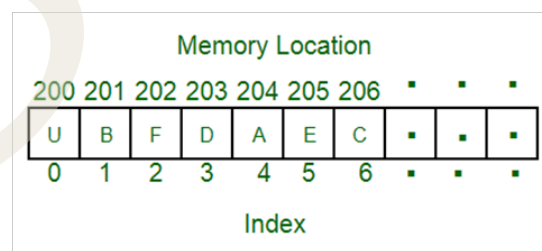


Fig 2.2.1 Array as a data structure

An array is characterized by three fundamental attributes, which are essential during its declaration: array name, array size, and array type.

Array Name : The array name serves as the identifier for the array. It is a user-defined

variable that allows for referencing and accessing individual elements within the array through their corresponding index values.

Array Size: The array size determines the maximum number of elements that the array can accommodate. It defines the range of index values that can be used to access the elements, typically from 0 to size-1.

Array Type

The array type specifies the data type of the elements stored within the array. This ensures that all elements in the array are of a uniform data type, such as integers, characters, or floating-point numbers.

For instance, consider an array named `arr` that is declared to hold 10 integer values. In this case, the array name is `arr`, the array size is 10, and the array type is `integer`, indicating that all elements within the array are of integer type.

2.2.2 Array Declaration

The declaration of an array is always specific to the programming language being used. In the C programming language, the syntax for declaring arrays is as follows:

Syntax:

```
Data_type array_name [n1][n2]...[nn];
```

Where `data_type` specifies the data type of elements in the array and `[n1][n2]...[nn]` are dimensions of the array. An array can be 1D, 2D or nD. For example

```
int arr [n1] ;
```

declare a 1D array where `n1` is size of the array

```
int arr [n1][n2] ;
```

is a 2D array where `n1` and `n2` specify the size of the array.

For example, consider a one dimensional array of integers with size 5. It can be declared as follows:

```
int arr[5];
```

When we declare an array using the above statement, the compiler will allocate a contiguous block of memory which is capable of storing five integer values. For example if 4 byte is the size required to store an integer in a memory location then 5*4 byte=20 byte are required to store the array.

2.2.3 One Dimensional Array

A one-dimensional array is a linear or sequential data structure used to store a collection of elements of the same data type in contiguous memory locations. Due to this continuous arrangement, each element can be efficiently accessed using its corresponding index

value.

In a one-dimensional array, the indexing starts from 0, meaning the first element is located at index 0. The last element of the array is located at index `array_size - 1`, where `array_size` represents the total number of elements in the array.

2.2.3.1 Declaration of 1D array

The declaration of a one-dimensional (1D) array follows a specific syntax in programming languages such as C. This syntax defines both the data type and the number of elements the array can store.

Syntax:

`Data_type array_name[number of elements];`

Where `Data_type` specifies the data type of elements in the array and number of elements specify the size of the array which is always a positive integer. For example the array **StudList** and **Mark** can be declared as follows

- ◆ `char StudList[5];`
- ◆ `int Mark[5];`

2.2.3.2 Initializing 1D array

Once an array is declared we need to insert data values into the array. It is called initialization of an array. There are 2 methods for initializing an array. They are discussed below.

Method 1: Compile time initialization

In this method the size of the array is specified along the name of the array inside a square bracket. All the data elements are specified inside a curly bracket by listing the elements using comma separated values. If the data values are characters then they are put inside a single inverted comma. For example consider initialization of arrays **StudList** and **Mark**.

- ◆ `StudList[5] = { 'A', 'B', 'C', 'D', 'E' };`
- ◆ `Mark[5] = { 75, 80, 65, 85, 70 };`

Method 2: Run time initialization

Using runtime initialization, users can get a chance of accepting or entering different values during different runs of a program. It is also used for initializing large arrays or an array with user specified values. In C programming language an array is initialized at runtime using `scanf()` function and is shown below.

```
//array declaration
```

```
char StudList[5];
```

```
//array initialization

for (i=0;i<5;i++)

{

scanf("%c" , &StudList[i]);

}
```

Now see using the same manner how we can initialize an integer array. It can be done as shown below.

```
// array declaration

int Mark[5];

// array initialization

int Mark[5];

for (i=0;i<5;i++)

{

scanf("%d" , &Mark [i]);

}
```

During initialization if the number of elements is less than the length of the array the remaining locations of the array are filled by the value '0'.

2.2.3.3 Accessing Elements from 1D array

To access an element of an array, you use the array name followed by the index number in square brackets. Indexing in arrays always starts from 0, meaning the first element is at index 0, the second at index 1, and so on. The last element is at index array_size - 1.

Syntax:

```
array_name [index];
```

Example:

Suppose we have the following array:

```
int Marks[5] = {85, 90, 78, 92, 88};
```

To access the first element, write:

```
Marks[0];
```


This will return the value 85.

To access the second element, write:

```
Marks[1];
```

This will return the value 90.

2.2.4 Memory representation of 1D array

Single-dimensional arrays are always allocated contiguous blocks of memory. This implies that all the elements in an array are always stored next to each other. For example let us see how the array Marks is stored in memory. The schematic representation is shown in figure 2.2.2.

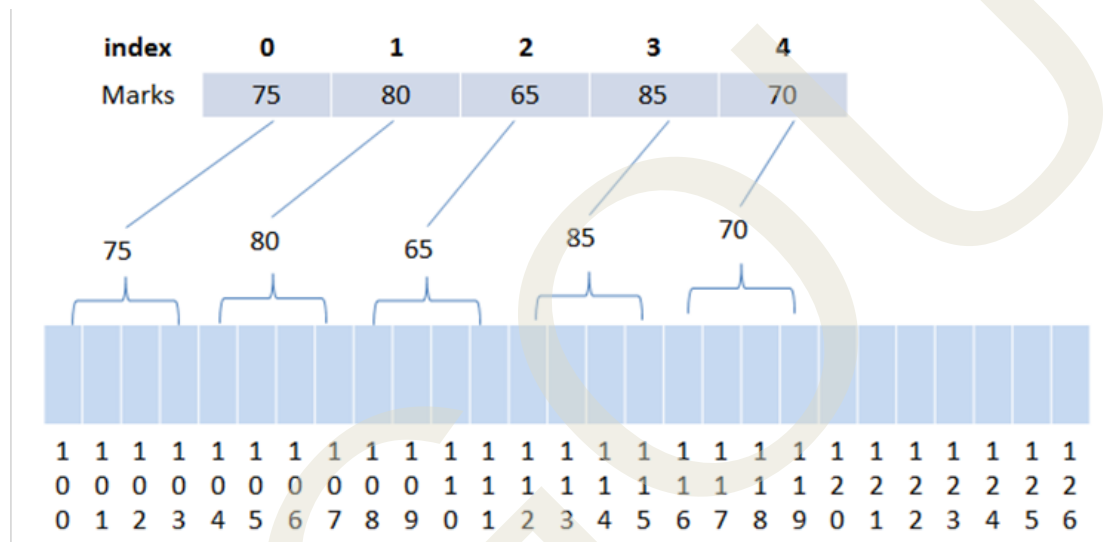


Fig 2.2.2 Schematic representation of 1D array in memory

Let the memory is byte addressable, that is one address location can hold one byte data. For simplicity assume the array address starts at memory location 100 and it is called base address of the array. In C programming we learned that data of type 'int' needs 4 bytes to store in memory. So as shown in the figure 2.2.2, memory location 100 to 103 to store 75, 104 to 107 to store 80, 108 to 111 to store 65, 112 to 115 to store 85 and 116 to 119 to store 70. Thus the array Marks occupies $4 \times 5 = 20$ contiguous bytes in memory and these bytes are reserved in the memory at the compile-time. Knowing the base address of the array, the address of memory location of each element can be calculated easily using the following equation.

Memory address of element, $array[i] = \text{Base address} + i \times (\text{size of data type of element in the array})$

Where i is the index of the element whose address wants to be calculated.

In the above example can you find out the address of the 2nd element? Let us see how it is calculated using the specified equation. Here the index or value of i is '2' and base address is 100. Also as mentioned above the size of data type is 4 as it is an integer. So we compute $Marks[2]$ as

$$\begin{aligned}
 \text{Mark}[2] &= 100 + 2 \times 4 \\
 &= 100 + 8 \\
 &= 108
 \end{aligned}$$

2.2.5 Two Dimensional Array

We learned that a one dimensional array is useful for representing linear lists. Suppose we want to represent a table or a matrix, in that case an array of arrays is used to store rows and columns. We can visualize it as a table as shown in the figure. 2.2.3. As you can see, there are 2 rows and 3 columns in the array A. The index values for row is 0 to 1 and index values for column is 0 to 3 clearly shown in the figure 2.2.3

	0	1	2
0	1	2	3
1	4	5	6

Fig 2.2.3 Two dimensional array A [2, 3]

2.2.5.1 Declaration of 2D Array using C

We have already understood the concept of a two-dimensional array and how it can be represented visually. Now, we will focus on how to declare a two-dimensional array in the C programming language. The general syntax for declaring a 2D array in C is as follows:

Syntax:

```
Data_type array_name[max_rows][max_columns];
```

In this syntax, max_rows defines the number of rows, while max_columns indicates the number of columns in the array. These values determine the dimensions of the array and specify how many elements it can hold across rows and columns.

2.2.5.2 Initialization of 2D Array

A two-dimensional array can be initialized in two primary ways: compile-time initialization and run-time initialization.

Method 1: Compile-Time Initialization

This approach involves assigning values to the array elements at the time of declaration. For example:

```
int A[2][3] = {1, 2, 3, 4, 5, 6};
```

In this example, the array A has 2 rows and 3 columns. The values within the braces are filled into the array from left to right, row by row. The first three values (1, 2, 3) will occupy the first row, and the next three values (4, 5, 6) will occupy the second row.

An alternative syntax for compile-time initialization uses nested braces to explicitly define each row:

```
int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

In this form, each inner set of braces represents one row of the array. Since the array has two rows, there are two sets of inner braces.

Method 2: Run-Time Initialization

In run-time initialization, the values are entered by the user during program execution. While a one-dimensional array uses a single for loop, a two-dimensional array requires two nested for loops, one for the rows and another for the columns. For example:

```
int x[2][3];

printf("Enter the elements:\n");

for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
        scanf("%d", &x[i][j]);
    }
}
```

2.2.6 Representation of Two-Dimensional Arrays in Memory

When discussing the representation of a two-dimensional array in memory, it is natural to question why such mapping is necessary. From the user's perspective, a 2D array appears as a matrix or table-like structure; however, this is an abstraction. In actuality, memory in a computer is linear, meaning it stores data in a one-dimensional sequence of memory locations. Therefore, even two-dimensional arrays are ultimately stored in a linear format in memory.

Two-dimensional arrays are often used to model structures like relational database tables or grids. While they appear to be arranged in rows and columns, the underlying memory structure uses a technique similar to that of a one-dimensional array. To store a 2D array in memory, it must be mapped into a linear format, which is typically done using either row-major order or column-major order, depending on the programming language or system architecture.

The total size of a two-dimensional array is determined by the product of the number of rows and the number of columns. For instance, an array with m rows and n columns will require $m \times n$ memory locations. This mapping process is essential to correctly access and manage elements stored in memory.

To illustrate the concept of organization and ordering consider a list of student names:

Nina, John, Alpha, Tony, and Smith. If these names are to be stored in alphabetical order in a 2D structure, the list would appear as follows:

Alpha	John	Nina	Smith	Tony
-------	------	------	-------	------

Fig 2.2.4 Array representation of student names

How can we arrange this in computer memory? One method we already studied is using arrays as in figure 2.2.1. But arrays have some limitations and disadvantages. Recall the limitations of array.

- ◆ The number of elements to be stored in an array should be known in advance.
- ◆ An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.
- ◆ Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
- ◆ Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

The data structure called linked list overcomes these limitations. So let us define what a linked list is. Linked List is a data structure used to store similar types of data in memory. It is a linear collection of data elements called nodes. Elements in a linked list are not stored in adjacent memory locations as in an array. They follow noncontiguous memory allocation.

A 3 X 3 two dimensional array is shown in the following figure 2.2.5. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.

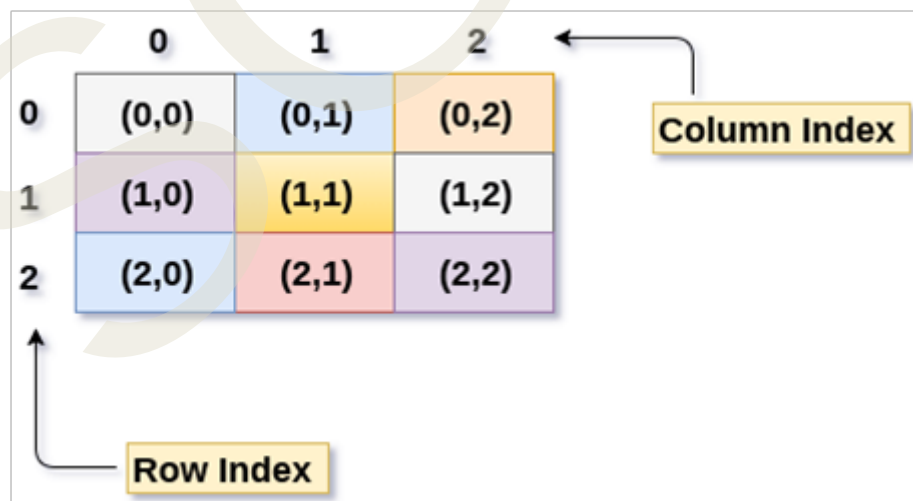


Fig 2.2.5 Visualization of 2D array

There are two main techniques of storing 2D array elements into memory. They are row major ordering and column major ordering.

2.2.6.1 Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

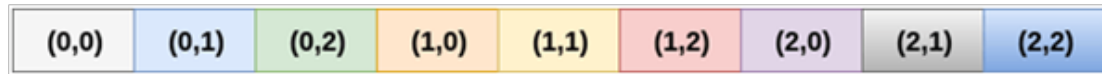


Fig 2.2.6. Memory allocation according to row major order

First, the 1st row of the array is stored into the memory completely, and then the 2nd row of the array is stored into the memory completely and so on till the last row as marked in figure 2.2.6.

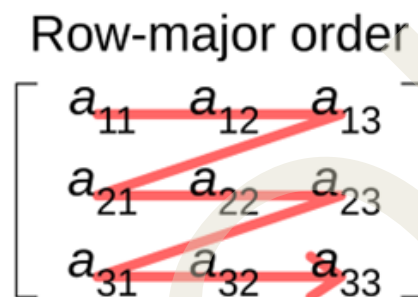
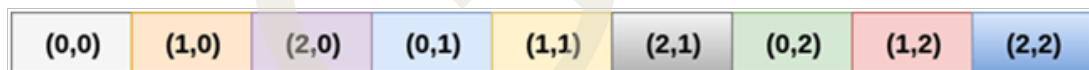


Fig 2.2.7. Row major ordering

2.2.6.2 Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in the image is given as follows.



Column-major order

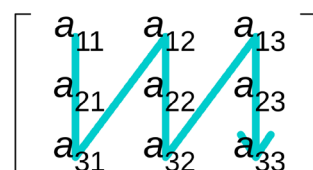


Fig 2.2.8 Column major ordering

First, the 1st column of the array is stored into the memory completely, then the 2nd column of the array is stored into the memory completely and so on till the last column of the array as shown in figure 2.2.8.

As in 1D array knowing the base address we can compute the address of any of the elements in the 2D array also. But it is necessary that the column size 'n' is given. For row major ordering the address of element at position (i, j) can be computed as follows

Address of $a[i][j] = B + ((i * n) + j) * \text{size}$

where B is the base address, n is the column size and i and j are indexes of the element whose address wants to be computed.

In column major implementation with starting index is '0', the address of any element is calculated using the equation given below

Address of $a[i][j] = B + ((j * m) + i) * \text{size}$

where m is the number of rows and B is the base address of the given 2D array.

2.2.7 Accessing of elements in 2D arrays

How to access 2D array elements? Using row index and column index we can access elements of a 2D array. For example we can access elements in the first row and second column of the array given below using $A[0][1]$.

	0	1	2
0	1	2	3
1	4	5	6

Similarly we can access all elements in a 2D array as

$A[0][0]=1$, $A[0][2]=3$ and so on.

2.2.7.1 Printing 2D array elements

You recall that 1D array elements can be printed using single for loop. But to print 2D array we use two nested for loops as shown in the program code below

```
int A[2][3] = {{1,2,3},{4,5,6}};
for (i=0; i<2; i++)
{
    for (j=0;j<3;j++)
    {
        printf("%d", A[i][j]);

    }
}
```

Output:

1 2 3

4 5 6

2.2.8 Advantages and disadvantages of array

Table 2.2.1 Advantages and disadvantages of array

Advantages	Disadvantages
Elements can be accessed easily using index numbers	Arrays are static; their size must be fixed at the time of declaration.
Contiguous memory allocation improves access speed and performance.	Memory wastage may occur if allocated size is not fully used
2D arrays can represent matrices for mathematical operations.	Only one data type can be stored (homogeneous structure).
Efficient for storing and processing multiple values of the same data type.	Cannot store mixed data types (e.g., int and char in the same array).

Recap

- ◆ Arrays store multiple elements of the same data type in contiguous memory locations.
- ◆ An array has three main attributes: its name, size, and type.
- ◆ Arrays can be one-dimensional, two-dimensional, or multi-dimensional depending on the number of indices used.
- ◆ In C programming, arrays are declared using specific syntax that includes the data type and size.

`Data_type array_name[size];`

- ◆ One-dimensional arrays can be initialized either at compile time or at runtime, and elements are accessed using index values starting from zero.
- ◆ The memory layout of a one-dimensional array is contiguous, and the address of each element can be calculated using the base address and the element's index.
- ◆ Two-dimensional arrays represent data in rows and columns and can be visualized as tables or matrices
- ◆ The memory representation of two-dimensional arrays can follow row-major or column-major ordering, affecting how elements are stored in memory.

Objective Type Questions

1. What specifies the maximum number of elements an array can hold?
2. Which index does array indexing start from?
3. What kind of data structure is an array?
4. How are elements stored in an array in memory?
5. What is the memory ordering technique where rows are stored one after another?
6. What is the memory ordering technique where columns are stored one after another?
7. Which loop is commonly used for initializing arrays at runtime?
8. What kind of array represents a matrix?
9. Arrays store elements of the same data type. What is this property called?
10. What is the output of the following piece of code?

```
#include<stdio.h>
int main()
{
    int mark[5] = {29, 30, 18, 17, 9};
    printf("%d", mark[1]);
    printf("%d", mark[4]);
}
```

Answers to Objective Type Questions

1. Size
2. Zero
3. Linear
4. Contiguous
5. Row-major
6. Column-major
7. for

8. Two-dimensional
9. Homogeneous
10. 10 and 9

Assignments

1. Explain the concept of arrays in C programming. Describe the three main attributes of an array with examples.
2. Write a C program to declare a one-dimensional array of integers, initialize it with values, and print all the elements using a loop.
3. Describe the difference between one-dimensional and two-dimensional arrays. Illustrate your answer with examples of declaration and initialization for both.
4. Write a C program to input elements into a 2D array of size 2x3 and display the elements in matrix form.
5. Explain row-major and column-major order in storing two-dimensional arrays in memory. Provide examples to illustrate both methods.

Reference

1. <https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-saman>

Suggested Reading

1. Sharma, A. K. “Data Structures using C”, 2e. Pearson Education India, 2013.
2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. “Data structures and algorithms”. Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
3. Weiss, Mark Allen. “Data structures and algorithm analysis”. Benjamin-Cummings Publishing Co., Inc., 1995.

Unit 3

Stack

Learning Outcomes

Upon completion of this unit, the learner will be able to

- ◆ explain the need for the stack as a Last In First Out data structure.
- ◆ introduce different operations on the stack.
- ◆ describe the concept of recursion.
- ◆ familiarise Polish notations.
- ◆ familiarise infix, prefix and postfix notations.

Prerequisites

In a busy canteen, clean plates are stacked one on top of the other. When someone wants a plate, they take the one from the top, and the next one becomes available. This continues until the stack of plates is empty. This real-life setup demonstrates the basic principle of a stack, where the last item placed is the first one removed.

In computing applications, a stack works in the same manner. That is a stack is a linear data structure in which elements of the same type are placed one above the other. In the stack, items are inserted and deleted from the top. To access elements of a data structure, we have to use some operations. Push() and Pop() are the two important operations used for inserting and deleting elements in a stack.



Fig 2.3.1 Stack of plates

Stacks are used in a similar way to handle tasks where the most recent item must be processed first. One major application of stacks is in function calls, especially recur-



sive functions, where each call is stored on the call stack and resolved in reverse order. Stacks are also useful in expression evaluation and conversion, such as converting infix expressions to postfix or evaluating postfix notation. In undo operations in text editors, every action is pushed onto a stack and can be undone in the reverse order it was performed. Backtracking algorithms like solving a maze or playing puzzles like Sudoku also use stacks to keep track of choices and reverse them when needed. In compiler design, stacks help in checking for balanced parentheses and maintaining the structure of code. These diverse applications make stacks an essential part of problem solving in programming and system design.

Key words

LIFO, Push, Pop, peek, Polish notations, Infix, prefix, postfix, FIFO, Enqueue, Dequeue

Discussion

2.3.1 Introduction to Stack

This unit introduces stacks, which are a fundamental type of linear data structure. A stack is an abstract data type used widely in real-world applications. It is organized in such a way that all insertions and deletions occur at one end, known as the top. Stacks can be implemented using either arrays or linked lists.

A stack is a basic structure used to store data, where the order of data matters. A common real-life example is how a roadside seller arranges books. Figure 2.3.3 shows the example of a stack operation, i.e. the books are stacked one on top of another, and to take one out, you must start from the top. If a new book is added, the one below it becomes inaccessible until the top book is removed. This method of operation means you remove the most recently added item first, and the earliest added item last. This is known as Last In First Out (LIFO). Whether using C, C++, Java, Python, or C# stacks can be implemented in various programming languages to handle such operations effectively.



Fig 2.3.2 Stack Operation, books Arrangements

If you remove all the items from a stack, you can access them in reverse chronological order. The recently added item will be the first item you remove from the stack, and the last item will be the first added item to the stack.

A stack is a collection of elements where items can only be added or removed at one end, known as the top of the stack. Stacks are also referred to as LIFO (last in, first out) structures.

2.3.2 Basic Stack Structure

The basic structure of a stack can be implemented in memory through multiple techniques. Figure 2.3.4 provides an example of how a stack might be organized.

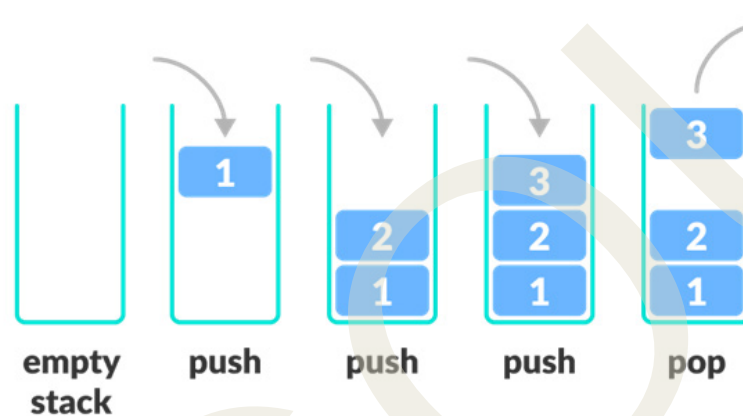


Fig 2.3.3 Visualization of the stack

2.3.1.2 Stack Representation

There are two other important ways to represent a stack: Using a one-dimensional array and a single linked list. Representation of stack is discussed in the following two sections.

1. Array Representation of stack

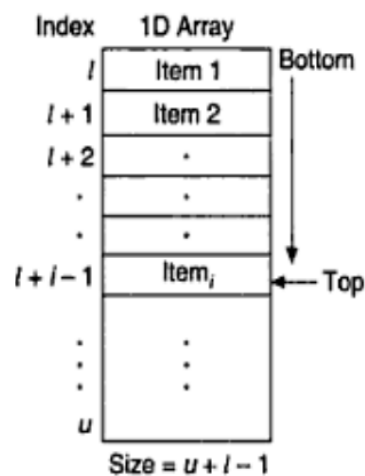


Fig 2.3.4 Array representation of the stack

2. Linked List Representation of stack

Using arrays to implement stacks is simple and convenient but limits the stack to a fixed size. However, in many cases, a stack's size may need to change while the program is running. To handle this, stacks can be implemented using linked lists. A singly linked list is sufficient for this purpose. In this structure, the **DATA** part holds the item, and the **LINK** part points to the next item in the stack. Figure 2.3.5 shows how a stack can be represented with a linked list. In this method, the first node represents the top of the stack, while the last node represents the bottom. When performing a PUSH operation, a new node is added at the beginning of the list, and a POP operation removes the node from the beginning.

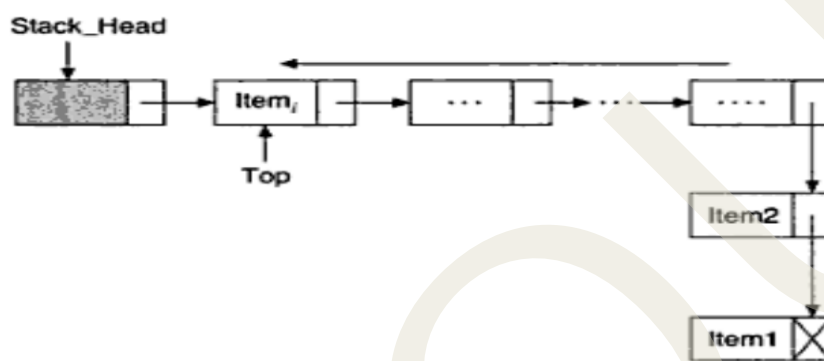


Fig 2.3.5 Linked list representation of the stack

2.3.1.3 Basic Operations of Stack

A stack is a type of linear data structure where elements are managed in a specific sequence, typically following the Last In, First Out (LIFO) principle. This means the most recently added item is the first one to be removed. The two main operations used in a stack are push and pop. The push operation inserts a new item onto the top of the stack. If the stack has reached its maximum capacity, it results in an overflow condition. The pop operation removes the item from the top of the stack. If there are no elements to remove, the stack is said to be in an underflow state.

Basic operations associated with stacks are:

1. **PUSH()**:- Used to insert an element into the stack
2. **POP()**:- Used to delete an element from the stack

To manage the stack effectively, it's important to monitor the status of its operations. For this reason, certain additional functions are included as part of stack operations.

- ◆ **Peek()**: The peek function allows viewing the top element of the stack without deleting it. It simply shows the item currently at the top.
- ◆ **isFull()**: This function checks if the stack has reached its maximum capacity.
- ◆ **isEmpty()**: This function checks if the stack has no elements left.

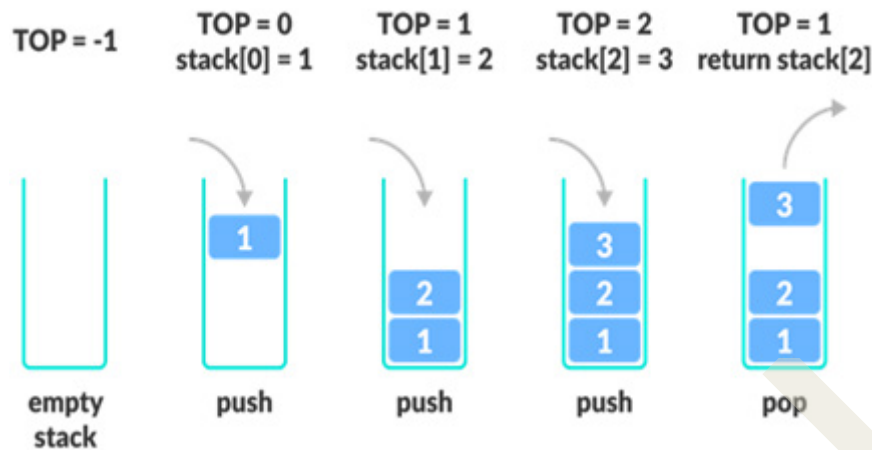


Fig 2.3.6 Working of stack operations

Figure 2.3.6 demonstrates how stack operations work in sequence. The TOP variable acts as a pointer that manages the elements in the stack. Initially, the stack is set up by assigning TOP the value -1. If TOP remains -1, it indicates that the stack is currently empty. When an item is pushed onto the stack, the TOP value increases by one, and the new item is inserted at the new TOP position. This process repeats as TOP takes on the values 0, 1, and 2, meaning that three elements have been successfully added to the stack.

To remove an element, the stack performs a POP operation. This action retrieves the item pointed to by TOP and then decreases the TOP value by one (in this case, from 2 to 1). As a result, the last inserted item, 'Item3', is removed from the stack. Through continued pop operations, all added elements can be removed in reverse order. Before executing a PUSH, it is important to verify that the stack is not already full. Similarly, before performing a POP, the stack should be checked to ensure it is not empty.

1. Push Operation

The push operation adds a new element to the stack. It inserts a data item at the top of the stack. The steps involved in performing the push operation are outlined below.

- Step 1: Verify whether the stack has reached its maximum capacity.
- Step 2: If the stack is already full, display an error message and stop the operation.
- Step 3: If there is space available, increase the TOP pointer to the next available position.
- Step 4: Insert the new data item at the position indicated by the updated TOP pointer
- Step 5: Confirm that the operation was completed successfully.

Refer Fig. 2.3.7 Working of push () operation.

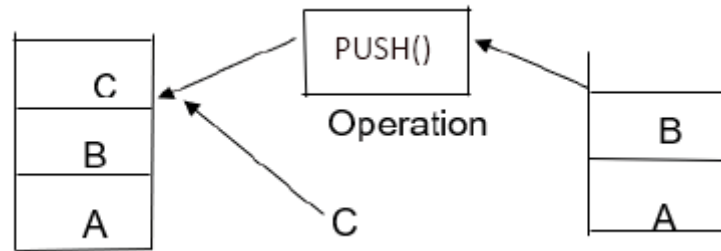


Fig 2.3.7 Working of push () operation

2. Pop Operation

The POP operation refers to taking out or retrieving the most recently added item from the stack. This action also frees up the memory used by that element. The pop() process includes the following steps.

Step 1: Verify whether the stack has no elements.

Step 2: If the stack is found to be empty, display an error message and terminate the process.

Step 3: If the stack contains elements, retrieve the data item located at the position indicated by the TOP pointer.

Step 4: Reduce the value of the TOP pointer by one.

Step 5: Indicate that the operation was completed successfully.

Refer Fig. 2.3.8 Working of pop () operation.

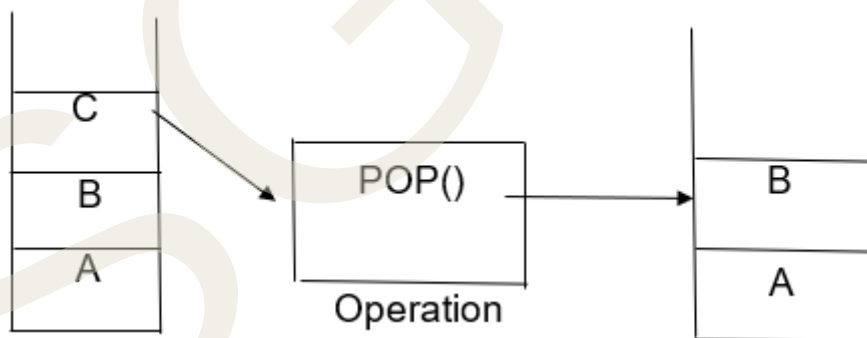


Fig 2.3.8 Working of pop () operation

3. Peek Operation

The peek operation is used to examine the top element of the stack without removing it. It helps in assessing the current state of the stack by displaying the most recently added item. This operation is useful when we need to know what's on the top of the stack without modifying its contents. By using peek, we can ensure that the stack is functioning correctly and that the push and pop operations are behaving as expected. It provides a quick way to access the top item and verify stack status during program execution.

2.3.1.4 Applications of Stack

Stacks have several important applications in computer science. Among these, four major uses are commonly highlighted. When a recursive function is executed, compilers rely on stacks to manage function calls internally. Stacks are also essential in evaluating mathematical expressions and verifying the correct placement of parentheses. Below are some key areas where stacks are significantly utilized.

- ◆ Recursion
- ◆ Arithmetic Expression Evaluation
- ◆ Expression Conversion
- ◆ Syntax Parsing
- ◆ Backtracking
- ◆ String Reversal

2.3.2 Recursion

One of the most effective ways to solve a complex problem is by addressing a smaller version of the same problem first. Recursion is a method used to approach a problem by simplifying it into a reduced version of itself and continuing this process with progressively smaller instances. The principle of recursion is based on the idea that certain problems can be solved more efficiently and with less effort when approached in this manner. This technique is widely used by mathematicians to tackle a range of mathematical problems, including calculating factorials, computing powers, and finding the greatest common divisor (GCD). In programming, recursion refers to the process where a function calls itself within its own definition.

There are two main types of recursion that can be used to solve problems: direct recursion and indirect recursion. Direct recursion is the more basic form, where a function directly calls itself in a single, straightforward step. This type of recursion involves only one function being invoked recursively. On the other hand, indirect recursion, also known as mutual recursion, occurs when a function calls another function, which eventually leads back to the original function being called again. This type of recursion involves multiple functions and steps in the recursive process.

2.3.2.1 Recursion Process in C language

Recursion is the process of repeating a set of instructions in a similar way as it was previously executed. The C programming language provides support for implementing recursion. To successfully execute recursion in C, two essential conditions must be satisfied. The first is the exit condition, and the second is the modification of the counter variable. The exit condition is crucial as it instructs the function when to stop its execution and return. If this condition is not defined by the programmer, the program may enter into an infinite loop, continuously calling the function without termination. The counter change condition ensures that some variable or value is updated with each recursive call, guiding the function toward its exit condition. In C, recursion can be implemented in two forms: tail recursion and non-tail recursion, both supported by the language.

1. Tail Recursion

Tail recursion occurs when a recursive function makes a call to itself and this call is the final operation performed within the function. It is a specific type of linear recursion. In tail recursion, no additional computation is performed after the recursive call returns; the result of the recursive call is typically returned directly. This type of recursion can often be converted into an iterative solution with ease. A classic example of a tail recursive function is demonstrated below.

Example: factorial of number 5

```
(factorial 5)
= (* 5 (factorial 4))
= (* 5 (* 4 (factorial 3)))
= (* 5 (* 4 (* 3 (factorial 2))))
= (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
= (* 5 (* 4 (* 3 (* 2 (* 1)))))
= (* 5 (* 4 (* 3 (* 2))))
= (* 5 (* 4 (* 6)))
= (* 5 (* 24))
= 120
```

Answer: 120.

Tail recursive methods are simple to transform into their iterative counterparts. Advanced compilers are capable of identifying tail recursion and automatically converting it into optimized iterative code. Tail recursion is commonly employed to create looping behavior in programming languages that do not provide explicit support for traditional loop constructs.

2. Non-Tail Recursion

In non-tail recursion, certain operations remain incomplete after the recursive call, meaning that the function performs additional tasks after returning from the recursive invocation. In such cases, the recursive function call is placed in the middle of the function or followed by further computation. Functions that follow this structure are referred to as non-tail recursive functions. An illustration of a non-tail recursive function is provided below:

Input number: 4

Output: Factorial is: 24

Explanation: $1 * 2 * 3 * 4 = 24$


The factorial of number 4 can be implemented as a non-tail recursive function,

Answer = 24

A recursive function has the potential to execute an infinite number of iterations, similar to a loop. To prevent a recursive function from running endlessly, it must satisfy two essential properties. These properties are centered around the concept of a defined stopping condition and a gradually progressing approach toward that condition.

Base criteria refer to a fundamental condition that must be present in every recursive function. It ensures that there is at least one specific situation or case where the function will not continue to call itself, thereby preventing infinite recursion. When this defined condition is satisfied, the recursive process terminates, allowing the function to return a result and halt further execution.

The recursive calls must advance in a way that each new call moves closer to meeting the base condition.

Table 1.4.1: Calculate $5!$ 

- 1 $5 = 5 * 4$
!
- 2 $4 = 4 * 3$
!
- 3 $3 = 3 * 2$
!
- 4 $2 = 2 * 1$
!
- 5 $1 = 1 * 0$
!
- 6 $0 = 1$
!



In the $5!$ calculation demonstrated in Table 1.4.1, each step simplifies the factorial into a smaller subproblem until it reaches the base case $0!$, which is directly defined as having a value of 1; at line 6, this value is determined immediately, not as the result of another factorial, and the process then retraces its steps from line 6 back to line 1.

Recursive functions offer both advantages and disadvantages, the most significant benefit lies in algorithm design, where recursion takes advantage of repetitive structures found in many problems, enabling clearer and more efficient algorithm descriptions, and making it particularly effective for defining objects with self-similar structures.

However, recursion also has limitations: it can reduce execution efficiency by placing excessive data on the run-time stack, and if the recursion depth becomes too large, it may cause stack overflow; in addition, some recursive functions redundantly recalculate values for the same parameters, resulting in unacceptably long runtimes even for relatively simple problems.

2.3.3 Tower of Hanoi Problem

The Tower of Hanoi serves as a classic example of solving non-numeric problems using recursion. This problem was created by French mathematician Edouard Lucas in the year 1883. The recursive method is applicable to a broad range of problems whose solutions naturally follow a recursive pattern.

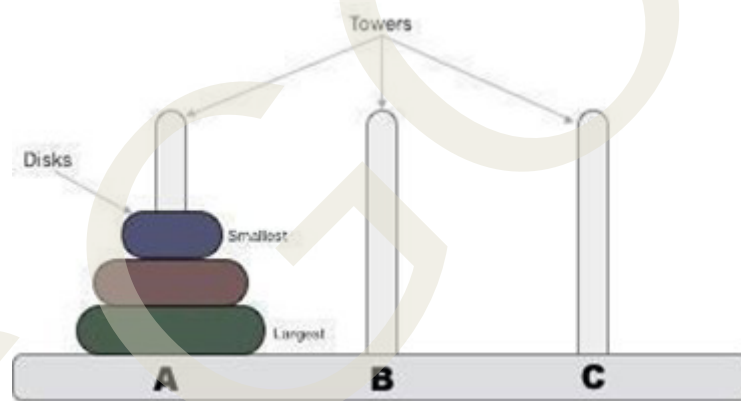


Fig 2.3.9 Tower of Hanoi

As illustrated in Figure 2.3.9, the Tower of Hanoi problem is a mathematical puzzle that involves three rods and n disks. The goal of the Tower of Hanoi puzzle is to transfer the complete stack of disks to a different rod while strictly following the given rules. These rules, which must be followed during the process, are outlined below.

1. A single disk is allowed to be moved at any given time.
2. Only the topmost disk on a stack can be taken off, meaning a disk may be moved only if it is the one at the top of the pile.
3. It is not permitted to position a larger disk above a smaller one in the stack.

Figure Fig 2.3.10 shows three rods labeled 1, 2, and 3, where rod 1 holds three rings of different sizes and colors, arranged with the largest ring at the bottom and the smallest

ring placed at the top. Rods 2 and 3 are initially empty, and the objective is to transfer all three rings from rod 1 to rod 3 by moving one ring at a time to another rod, either if that rod is empty or has a larger ring on top. To solve the Tower of Hanoi puzzle with three disks, the entire stack must be moved from rod 1 to rod 3 while strictly following the rules stated earlier.

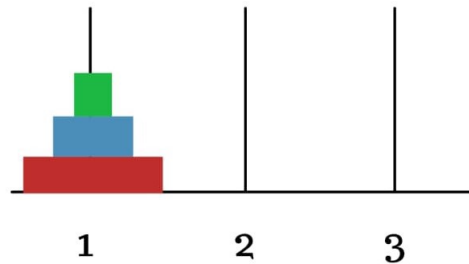


Fig 2.3.10 Illustration for Three Disks

Step 1: The smallest green disk, the uppermost disk on the stack, is shifted from rod 1 to rod 3.

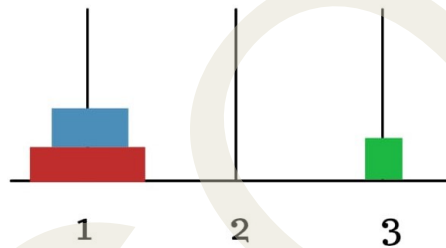


Fig 2.3.11 Illustration of Step

Step 2: Next, the uppermost disk on rod 1 is the blue-coloured disk shifted to rod 2.

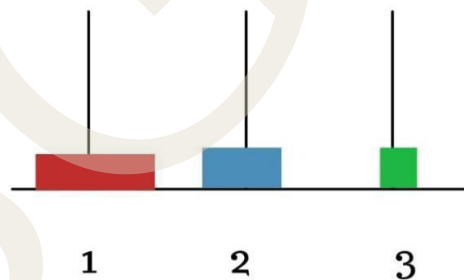


Fig 2.3.12 Illustration of Step 2

Step 3: The smallest disk placed on rod 3 is shifted back to the top of rod 2.

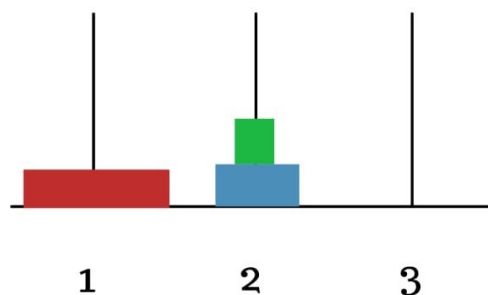


Fig 2.3.13 Illustration of Step 3

Step 4: Now, the largest red disk is allowed to be shifted from rod 1 to its destination, rod 3.

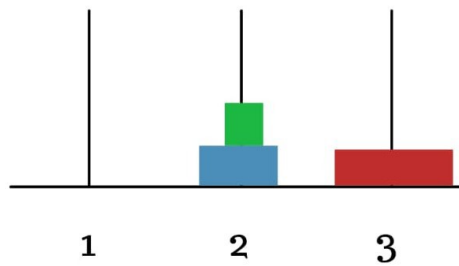


Fig 2.3.14 Illustration of Step 4

Step 5: At this stage, the two disks currently placed on rod 2 must be moved to their final destination, which is rod 3, where they will be stacked above the red disk.

Therefore, to begin the process, the smallest green disk that is positioned on top of the blue rod needs to be transferred to rod 1.

This step is necessary to make space for the other disks to be moved correctly onto the destination rod while maintaining the stack order.

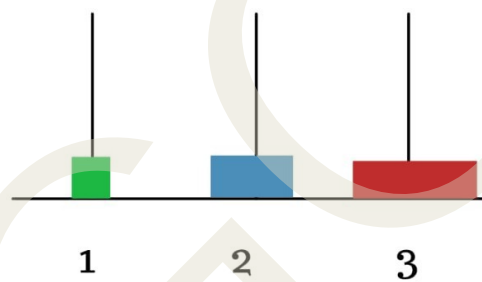


Fig 2.3.15 Illustration of Step 5

Step 6: Next, the blue disk is permitted to be shifted to its destination rod 3 stacked on to the top of the red disk.

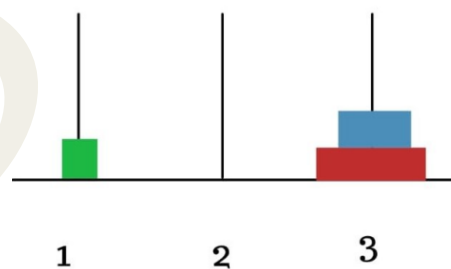


Fig 2.3.16 Illustration of Step 6

Step 7: Finally, the smallest green-coloured rod is also shifted to rod 3, which would now be the uppermost rod on the stack. So, the tower of Hanoi for three disks has been solved.

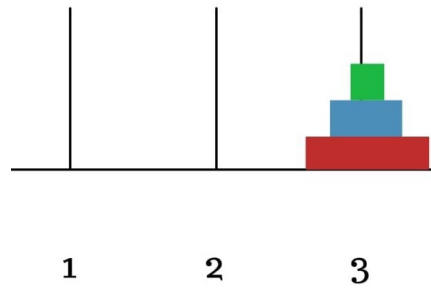


Fig 2.3.17 Illustration of Step 7

2.3.4 Evaluation of Arithmetic Expression

An arithmetic expression is made up of both operands and operators. Operands can either be constants or variables, while operators come in different types. These operators perform specific functions within the expression.

- ◆ Addition (+), subtraction (-), Multiplication (*), division (/), Exponentiation (^), modulo (%).
- ◆ Relational operators (<, >, <=, >= etc)
- ◆ Boolean operator (AND, OR, NOT, XOR, etc)

A simple arithmetic expression is given below:

$$A+B*C-E^F$$

The problem with evaluating the given expression is the order of evaluation.

A solution is to assign both precedence and associativity to each operator. For instance, Table 2.3.2 provides a group of common operators along with their respective precedence and associativity. In the table, a precedence level of 1 indicates the highest precedence, whereas a level of 5 signifies the lowest precedence.

Table 2.3.2 Precedence and associativity of operators

Precedence	Operator	Description	Associativity
1	()	Parentheses	Left-to-Right
	[]	Square bracket	
2	^	Exponentiation	Right-to-Left
3	*, /, %	Multiplication, division, modulus	Left-to-Right
4	+/-	Addition, subtraction	Left-to-Right
5	<, <=, >, >=	Relational operators	Left-to-Right

We can evaluate the expression $A + B * C - E \wedge F$ as:

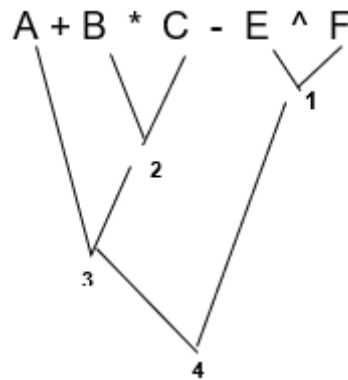


Fig 2.3.18 Evaluation of Arithmetic Expression

The above expression will be executed in the sequence of 1,2,3,4

It is important to understand that the rules for operator precedence and associativity mentioned above may differ depending on the programming language used. Additionally, the method described for evaluating expressions is not efficient, as it requires scanning the expression multiple times. This inefficiency can be addressed effectively by following the steps outlined below.

1. Conversion of a given expression into special notations.
2. Evaluation of an object code using stack.

2.3.4.1 Polish Notation

Polish Notation refers to the method of placing the operators in an expression either before or after their operands. This technique involves altering the usual position of operators relative to the operands in an arithmetic expression. There are three distinct notations commonly used to represent arithmetic expressions using this method.

- ◆ Infix notation
- ◆ Prefix notation
- ◆ Postfix notation

1. Infix notation

Infix notation, also known as infix expressions, is a widely used form of mathematical notation. In this format, the operator is positioned between its two operands. For instance, if A and B are two operands with values 2 and 3 respectively, the expression $A+B$ represents an infix expression where the operator “+” appears between A and B; this can also be written as $2+3$, clearly showing why the format is termed infix, because the operator is placed between the operands.

Syntax:

<operand> <operator> <operand>.

2. Prefix notation (Polish Notation)

Prefix notation specifies that an operator must appear before its corresponding operands. This form of notation is also referred to as **Polish notation**. For instance, consider the expression $A+B$, where A and B are operands and $+$ is the operator; in prefix form, this becomes $+AB$, meaning the operator is placed before the operands, hence the term “prefix.”.

Syntax:

$\langle \text{operator} \rangle \langle \text{operand} \rangle \langle \text{operand} \rangle$

3. Postfix notation (Reverse Polish Notation)

Postfix notation specifies that the operator must appear as a suffix following the operands. This format is also referred to as **reverse Polish notation**. For instance, consider the expression $A+B$, where A and B are operands and $+$ is the operator; the postfix equivalent of this expression is $AB+$, with the operator placed after the operands, hence the name postfix.

Syntax:

$\langle \text{operand} \rangle \langle \text{operand} \rangle \langle \text{operator} \rangle$

Some additional examples of infix, prefix and postfix expressions are shown below:

Infix Notation	Prefix Notation	Postfix Notation
$A+B$	$+AB$	$AB+$
$(A-C)*B$	$*-ACB$	$AC-B*$
$A+(B*C)$	$+A*BC$	$ABC*+$

2.3.4.2 Conversion of infix to postfix expression

An infix expression refers to an expression where the operator is placed between the operands. For instance, the expression $a + b / c - d$ is written in infix form. To convert an infix expression into a postfix expression, we can follow a specific set of rules.

- ◆ **Step 1:** Traverse the expression starting from the leftmost character and move sequentially to the right.
- ◆ **Step 2:** Each time you find an operand in the expression, insert it into the stack using the push operation.
- ◆ **Step 3:** If the stack has no elements or if the topmost item is a left parenthesis, then insert the current operator onto the stack.
- ◆ **Step 4:** When the symbol ‘(’ appears in the input, push it directly onto the stack.
- ◆ **Step 5:** If a ‘)’ symbol is encountered, continue to pop elements from the stack and display the operators until a left parenthesis ‘(’ is found.

- ♦ **Step 6:** If the operator being read has a higher precedence than the operator at the top of the stack, then push the incoming operator onto the stack.
- ♦ **Step 7:** If the incoming operator has lower precedence than the stack's top operator, pop the top operator, display it, and then compare the incoming operator with the new top operator on the stack.
- ♦ **Step 8:** In case the precedence of the incoming operator and the top of the stack is equal, determine the correct action based on associativity rules.

If the associativity rule is left-to-right, first pop and display the top operator, then push the incoming operator.

If the associativity is right-to-left, directly push the incoming operator onto the stack.

Example: Convert the infix notation into postfix notation

The infix notation is $A+B*C/(E-F)$

Move	Input string	Output stack	Output
1	$A+B*C/(E-F)$		A
2	$A+B*C/(E-F)$	+	A
3	$A+B*C/(E-F)$	+	AB
4	$A+B*C/(E-F)$	+	AB
5	$A+B*C/(E-F)$	+	ABC
6	$A+B*C/(E-F)$	+/	ABC*
7	$A+B*C/(E-F)$	+/ (ABC*
8	$A+B*C/(E-F)$	+/ (ABC*E
9	$A+B*C/(E-F)$	+/ (-	ABC*E
10	$A+B*C/(E-F)$	+/ (-	ABC*EF
11	$A+B*C/(E-F)$	+/	ABC*EF-
12	$A+B*C/(E-F)$		ABC*EF-/+

The result is $ABC*EF-/+$

2.3.4.3 Conversion of Infix to Prefix Expression

An infix expression refers to a mathematical expression in which the operators are written between the operands. For example, in the arithmetic expression $a + b / c - d$, the operations follow a specific sequence based on operator precedence. First, the division operation " b / c " is performed. The result is then added to the operand " a ", and finally, the operand " d " is subtracted from the intermediate result. This final computed value represents the solution to the arithmetic expression " $a + b / c - d$ ". Such infix expressions can be converted into either prefix or postfix notations using stack-based methods. Consider the expression " $a + b$ " in infix form, where " a " is the first operand, " $+$ " is the operator, and " b " is the second operand. This same expression in prefix

notation becomes “+ a b”, where the operator “+” comes first, followed by the operands “a” and “b” respectively.

Step 1: Begin by reversing the given infix expression.

Step 2: Traverse the characters of the reversed expression one at a time.

Step 3: If a character is an operand, transfer it directly to the prefix output.

Step 4: If the character is a closing parenthesis, push it onto the stack.

Step 5: If the character is an opening parenthesis, remove elements from the stack until the matching closing parenthesis is found.

Step 6: If the character is an operator:

- ◆ If its precedence is higher than or equal to the operator at the top of the stack, push it onto the stack.
- ◆ If its precedence is lower than that of the top stack operator, pop operators from the stack and add them to the prefix output until the condition is satisfied, then repeat the check with the new top of the stack.

Step 7: Once all characters have been processed, reverse the resulting prefix output to obtain the final prefix expression.

Example: Convert the infix expression into prefix notation.

The infix notation is $(P+(Q*R)/(S-T))$

Move	Symbol Scanned	Stack	Output
1))	-
2)))	-
3	T))	T
4	-))-	T
5	S))-	TS
6	()	TS-
7	/)/	TS-
8))/)	TS-
9	R)/)	TS-R
10	*)/)*	TS-R
11	Q)/)*	TS-RQ
12	()/	TS-RQ*
13	+)+	TS-RQ*/
14	P)+	TS-RQ*/P
15	(Empty	TS-RQ*/P+

Reverse the obtained expression TS-RQ*/P+.

The final result is that the prefix expression is + P/*QR-ST.

Some of the additional examples of converting infix to postfix notations and infix to prefix notations are given below. Understand each of these examples and do these conversions on your own.

Example of infix to postfix notation conversion:

No	Infix	Postfix
1	A+B	AB+
2	A+B-C	AB+C-
3	(A+B)*(C-D)	AB+CD-*
4	A*B/C	AB*C/
5	2+3*4	234*+
6	A*(B+C)/D-G	ABC+*D/G-

Example of infix to prefix notation conversion:

No	Infix	Prefix
1	A+B	+AB
2	A+B-C	-+ABC
3	(A+B)*(C-D)	*+AB-CD
4	A/B*C-D+E/F/(G+H)	+-*//ABCD//EF+GH
5	((A+B)*C-(D-E))*(F+G)	*-*+ABC-DE+FG
6	A-B/(C*D/E)	-A/B/*CDE

2.3.4.4 Evaluation of Postfix Expression

In infix expressions, it becomes challenging for a machine to manage and identify the priority and significance of operators within the expression. On the other hand, postfix expressions clearly define the priority and importance of operators, making them easier for a machine to process. As a result, executing a postfix expression is simpler and more efficient for a machine compared to an infix expression. Postfix notation is commonly used to represent algebraic expressions due to its clarity in operator precedence. Expressions written in postfix form are evaluated more quickly than those in infix form. The procedure for evaluating postfix expressions is outlined below:

Step 1: As you read the expression from left to right, if the current element is an operand, insert (push) it into the stack.

Step 2: If the element encountered is an operator (excluding the NOT operator), remove (pop) the top two operands from the stack for evaluation.

Step 3: If the element is a NOT operator, then remove (pop) only one operand from the stack and perform the evaluation on that single operand.

Step 4: Once the evaluation is complete, insert (push) the result back into the stack.

Step 5: Continue this process until the entire expression has been read and processed from the stack.

Algorithm for Evaluation of Postfix Expression

Step 1: Begin by scanning the postfix expression from left to right.

Step 2: On the first iteration, read the initial element in the expression.

Step 3: If the element encountered is an operand, then:

Push the operand onto the stack for temporary storage.

Step 4: If the element is an operator, then:

Pop two operands from the top of the stack.

Perform the operation using the operator on the two popped operands.

Push the resulting value back onto the stack.

Step 5: If all elements in the expression have been processed, then pop the final result from the stack and consider it as the output.

If elements are still remaining, repeat the process by returning to Step 1.

For example, evaluate the postfix expression:

$AB+C*D/$

Where $A=2$, $B=3$, $C=4$, and $D=5$ starting from left to right.

1. The first element is an operand A, PUSH A, into the stack.

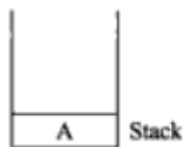


Fig 2.3.19 illustration of step 1

2. The second element is operand B, and PUSH B is also added to the stack.

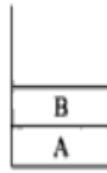


Fig 2.3.20 illustration of step 2

3. The third element, “+”, is an operator, POP two elements from the stack. Then, evaluate A and B.

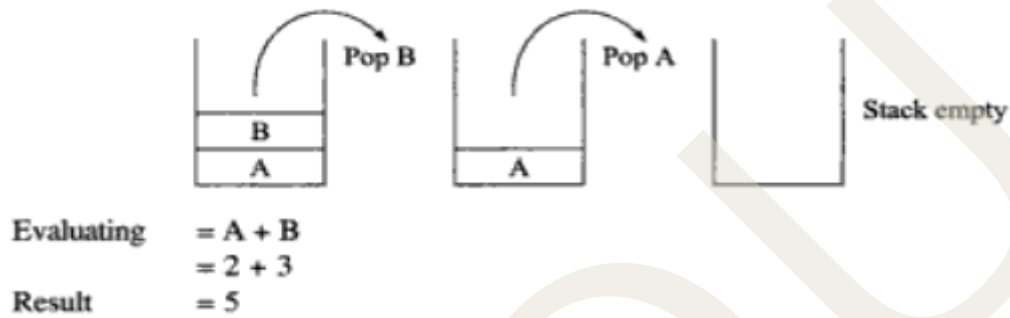


Fig 2.3.21 illustration of step 3

4. PUSH the result, element five, into the stack.



Fig 2.3.22 illustration of step 4

5. The next element, C, is an operand; PUSH it into the stack.

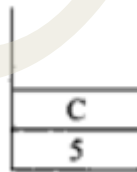


Fig 2.3.23 illustration of step 5

6. Next, “*” is an operator, POP two operands from the stack, then evaluate elements five and C.

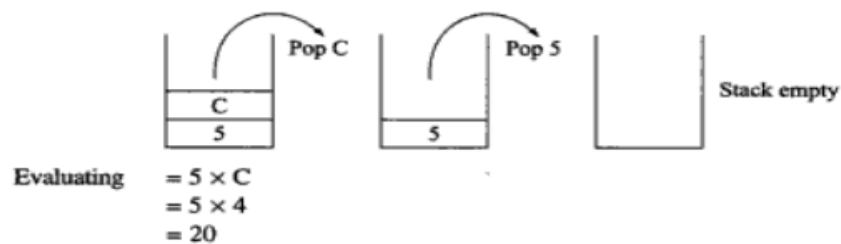


Fig 2.3.24 illustration of step 6

7. PUSH the result 20 into the stack.

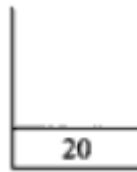


Fig 2.3.25 illustration of step 7

8. Next D is an operand; PUSH it into the stack.

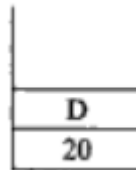


Fig 2.3.26 illustration of step 8

9. Next, “/” is an operator.

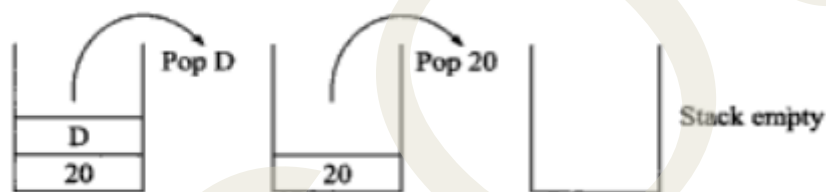


Fig 2.3.27 illustration of step 9

POP two operands from the stack and evaluate 20 and D.

Evaluating

$$20/D = 20/5 = 4$$

End of expression. Finally, the result is value 4.

Recap

- ◆ A stack is a linear data structure that follows the Last In First Out (LIFO) principle.
- ◆ The most recently added element is removed first in a stack.
- ◆ Real life examples include a pile of plates or books where only the top item is accessible.
- ◆ The two main stack operations are push (insert) and pop (remove).
- ◆ Stacks are widely used to manage function calls in computer programs.
- ◆ Recursive function calls are handled using the system's call stack.
- ◆ The stack stores local variables and return addresses during recursive calls.

- ◆ Undo and redo features in software applications are implemented using stacks.
- ◆ Expression conversion such as infix to postfix or prefix uses stack operations.
- ◆ Postfix expression evaluation uses a stack to compute results efficiently.
- ◆ Stacks are used in syntax checking for balanced parentheses in code.
- ◆ They are also used in parsing mathematical expressions in compilers.
- ◆ Tower of Hanoi, a classic recursive problem, is solved using stack logic.
- ◆ Stack helps in reversing strings or lists in programming tasks.
- ◆ Stack can be implemented using arrays or linked lists.
- ◆ Backtracking problems like maze solving use stacks to store paths.
- ◆ Web browsers use stacks to handle back and forward navigation of pages.
- ◆ Depth First Search (DFS) in graph traversal uses an explicit or implicit stack.
- ◆ Polish notation and Reverse Polish Notation are stack-based expression formats.
- ◆ Stacks are essential for managing memory and function flow in programming.

Objective Type Questions

1. What type of data structure is a stack?
2. Which principle does a stack follow?
3. Which end of the stack allows insertions and deletions?
4. What is the condition of an empty stack using arrays?
5. What is the condition of a full stack using arrays?
6. Which pointer is used to manage stack operations?
7. Which operation adds an element to the stack?
8. Which operation removes the top element of the stack?
9. What error occurs when trying to push to a full stack?
10. What error occurs when trying to pop from an empty stack?
11. Which operation displays the top element without removing it?

12. Which data structure is used for dynamic stack implementation?
13. Which method is used to reverse a string using stack?
14. What is the result of postfix expression evaluation?
15. Which type of notation places the operator before operands?
16. Which notation is easier for machines to evaluate?
17. Which recursion type calls itself as the last operation?
18. What is the base condition in recursion?
19. What is the output of factorial(4)?
20. Which classic problem is solved using recursion and three rods?

Answers to Objective Type Questions

1. Linear
2. LIFO
3. Top
4. $\text{top} < l$
5. $\text{top} \geq u$
6. Top
7. Push
8. Pop
9. Overflow
10. Underflow
11. Peek
12. LinkedList
13. Backtracking

- 14. Operand
- 15. Prefix
- 16. Postfix
- 17. Tail
- 18. Termination
- 19. 24
- 20. Hanoi

Assignments

1. Explain the Array and Linked List representations of a stack. Include diagrams and discuss the advantages and disadvantages of each method.
2. Describe the basic operations of a stack (PUSH, POP, PEEK). Write algorithmic steps and provide examples to illustrate each operation.
3. Discuss the role of stacks in arithmetic expression evaluation. Explain infix, prefix, and postfix notations with conversion examples and operator precedence rules.
4. What is recursion? Explain tail and non-tail recursion with suitable examples. Also, describe the importance of the base condition and progressive approach.
5. Explain the Tower of Hanoi problem using recursion. Describe the steps involved in solving it for 3 disks and provide diagrams for each move.
6. Write an algorithm to evaluate a postfix expression using stack. Demonstrate the evaluation process with a suitable example including all intermediate stack states.
7. Transform the following expression to prefix and postfix.
$$P+Q-Z$$
8. Transform the following expression from prefix to infix.
$$+-ABC$$
9. What is the postfix expression for the infix expression?
$$a-b-c$$

10. What is the postfix expression for the following infix expression?

$$a/b^c-d$$

11. What is the corresponding postfix expression for the given infix expression?

$$a*(b+c)/d$$

12. What are all the primitive operations in the stack?

13. Convert the following infix expression into a postfix expression.

$$(A+B)*C$$

$$(A+B)*C/E$$

$$((A-(B+C))*D)/(E+F)$$

14. Evaluate the following postfix expressions

$$AB+C-BA+C/-$$

$$ABC+*CBA-+*$$

Where $A=1$, $B=2$, $C=3$.

Suggested Reading

1. Sharma, A. K. "Data Structures using C", 2e. Pearson Education India, 2013.
2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and Algorithms". Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
3. Weiss, Mark Allen. "Data structures and algorithm analysis". Benjamin-Cummings Publishing Co., Inc., 1995.
4. <https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-samantha.pdf>



Unit 4

Queue

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- ◆ define the basic concepts and structure of a queue data structure, including the roles of front and rear pointers.
- ◆ list the basic operations of a queue such as enqueue, dequeue, peek, isFull, and isEmpty.
- ◆ describe how enqueue inserts elements at the rear and dequeue removes elements from the front following the FIFO principle.
- ◆ explain the working of supporting functions like peek, isFull, and isEmpty in managing the queue.
- ◆ classify the different types of queues such as simple queue, circular queue, double-ended queue, and priority queue based on their structure and operations.

Prerequisites

At a railway reservation counter, people stand in a line waiting for their turn to book tickets. The person who arrives first is served first, and others wait in order. This is a real-life example of a queue. It shows how tasks are handled fairly, one after the other. A queue is a linear data structure where elements are added at the rear and removed from the front. It works on the principle of First in First Out (FIFO). This means the first element that goes in is the first one that comes out. Queues are used in many areas of computer science such as operating systems, networking, data buffering, and task scheduling. They help in handling tasks in the right order and improve the performance of programs. Without queues, many real-world and digital systems would not work properly.

Learning queues offers several important advantages for learners in the field of computer science. It helps in managing data in an organized and efficient way, which is essential for building real-time applications such as CPU scheduling, print spooling, and network data handling. By studying queues, learners develop strong problem-solving skills and logical thinking, as they learn to structure operations in a systematic order. Queues also form the foundation for understanding more advanced topics like graph traversal, tree



algorithms, and operating system process management. As a result, learners become better equipped to write clean, efficient programs and handle real-world computing problems. Mastery of queues also makes it easier to grasp other key data structures and algorithms, which are vital in software development, coding competitions, and technical interviews.

Key words

Rear, Front, Enqueue, Dequeue, Priority Queue, Circular Queue, Double-Ended Queue (Deque)

Discussion

2.4.1 Introduction to Queue

The queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The deletion end of the queue is called the front end, and the insertion end is called the rear end. The first element in a queue will be the first to be removed from the list. Therefore, the queues are also called FIFO (First in First Out) lists. Let's look at a real-life example; A queue of people standing in line to enter a venue. A new person arriving would join the end of the line, while the person at the front would leave the line and enter the venue.

Definition: A queue is an ordered list where insertions are done at one end (rear), and deletions are done at the other end (front). The first element to be inserted in the queue is the first to be deleted from the queue. Hence, it is called the First in First out (FIFO) or Last in Last out (LILO) list.



Fig 2.4.1 FIFO Operation in Queue

An important example of a queue in a computer system occurs in a time-sharing system. The programs with the same priority form a queue, and each program in the queue will execute first in the first out manner. Operating systems also use several different queues to control processes within a computer. The scheduling of program execution is typically based on a queuing algorithm that tries to execute programs as quickly as possible.

The queue data types have the following operations: first initialize the queue and check whether the queue is empty or full. If the queue is empty, insert a new element into the queue and continue the insertion process until the queue is full. On the other hand, if the queue is full, retrieve the first element and delete the element using the First in First out

(FIFO) method. The queue is a data structure somewhat similar to Stacks. The structure of a queue is open at both its ends. One end of the queue is always used to insert data called enqueue, and the other is used to remove data called dequeue. The queue process follows the First-In-First-Out data managing methodology.

Figure 2.4.2 represents a real-world example of a queue. The Figure is a ticket window counter queue; all three people are standing in the queue in a first-in, first-out manner. The first entered person will be getting the first ticket from the counter. The person who comes second gets the ticket second. So, the person who comes last gets the tickets last.

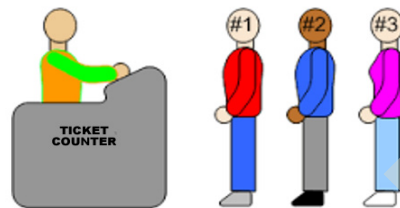


Fig 2.4.2 Ticket Window Counter Queue

2.4.2 Basic Operations in Queue

The queue data structure can also be implemented using arrays, linked lists, pointers and structures. For the sake of simplicity, the queue data structure can be implemented using a one-dimensional array. There are two important basic operations in the queue data structure. They are enqueue and dequeue.

1. enqueue operation - The enqueue is the process of inserting an element into the queue. In the queue, elements are always inserted at the rear end of the queue.
2. dequeue operation - The dequeue is the process of removing an element from the queue.

Elements from the queue are always removed from the front end of the queue.: To perform queue operations efficiently, some additional helper functions are commonly used. These functions assist in checking the state of the queue or retrieving values without modifying the structure.

◆ peek()

This function helps to see the data at the front end of the queue, and this function gets the element at the front end of the queue without removing it.

◆ isfull()

This function checks if the queue data structure is full. We use a single dimension array to implement a queue; we check for the rear pointer to reach MAXSIZE to determine that the queue is full.

◆ isempty()

This function checks if the queue is empty or not. It returns true if the queue contains a zero number of elements means the queue is empty.

2.4.2.1 Enqueue Operation in Queue

The enqueue operation in the queue is performed in the rear end of the queue. The data insertion process cannot perform if the queue is full. The queues maintain two data pointers, front and rear. Therefore, queue data structure operations are comparatively more difficult to implement than a stack data structure. When the queue has space, the enqueue operation inserts an element and updates the queue 'rear' pointer by incrementing $\text{rear} = \text{rear} + 1$.

The following steps are used to insert data elements into a queue.

Step 1: Check if the queue is full.

Step 2: If the queue is full, produce an overflow error and exit.

Step 3: If the queue is not full, increment the rear pointer to the next empty space.

Step 4: Add a data element to the queue location, where the rear is pointing.

Step 5: Return success.

Figure 2.4.3 represents the enqueue operation in the queue. The queue data structure contained 44, 55 and 66 data elements in the first, second and third memory location, respectively. The current status of a queue is $Q[1]=44$, $Q[2]=55$ and $Q[3]=66$.



Fig 2.4.3 Operation on Queue

The rear end value of the queue is 4. If we are going to insert a fourth data element 77 into the queue at the rear-end. Then the queue's new status contains four data elements, and the newly added element is $Q[4] = 77$, which is shown in Figure 2.4.4.

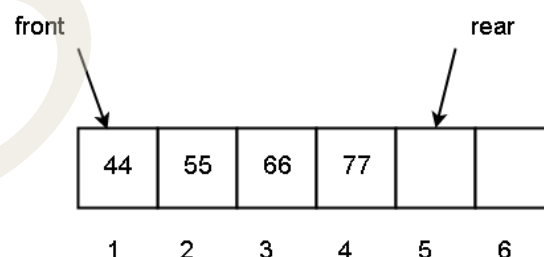


Figure 2.4.4 After Enqueue Operation on Queue

2.4.2.2 Dequeue Operation in queue

The dequeue operation removes elements from the front end of the queue, but the queue must not be empty for dequeue operation. Accessing data from the queue is a process of

two tasks: first, access the data where the front end is pointing and second, remove the data after access. In the dequeue operation, elements from the front end are removed, and the 'front' pointer is updated. The following steps are used to perform the dequeue operation:

Step 1: First, check if the queue is empty.

Step 2: If the queue is empty, produce an underflow error and exit.

Step 3: If the queue is not empty, access the data where the front is pointing.

Step 4: Increment front pointer to point to the next available data element.

Step 5: Return success.

Figure 2.4.5 represents the dequeue operation in the queue. The queue data structure contained three data elements 34, 15, and 54, in the first, second, and third memory locations. In the queue data elements, the dequeue operation removes elements from the front end of the queue.

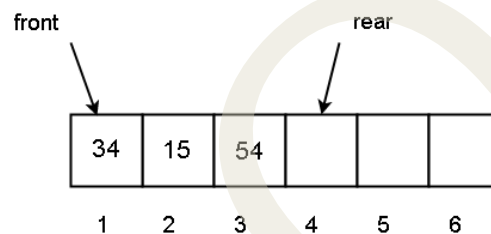


Fig 2.4.5 Dequeue Operation on Queue

The first memory location of the queue contains value 34, that is, $Q[1] = 34$, and the front pointer points to value 1. The element $Q[1] = 34$ is deleted in the queue in the first dequeue operations. After deletion of the first element the front pointer is incremented to ' $1 + 1 = 2$ '. It focuses on the second element in the queue $Q[2] = 15$. After the first dequeue operation, the remaining queue data elements are shown in Figure 2.4.6. Repeating the dequeue operation, all the inserted data elements are removed from the queue, and the queue will become empty.

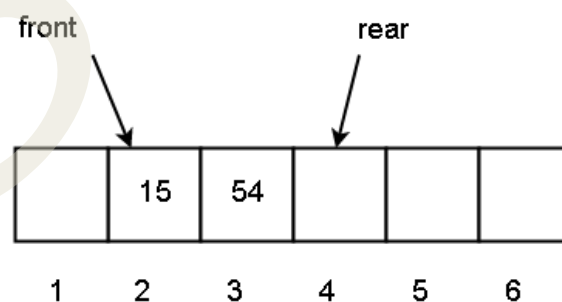


Fig 2.4.6 After Dequeue Operation on Queue

Understand the basic operations like enqueue and dequeue in the queues data structure; some supportive functions are used in the queue to improve the efficiency of the operations. The supporting functions of a queue are the peek(), isfull() and isempty()

functions. The `peek()` function always returns the head of the queue. The `isfull()` function checks for the rear pointer to reach `MAXSIZE` to determine that the queue is full. If the front value is less than `MIN` or `0`, the `isempty()` function tells that the queue is not yet initialized.

2.4.3 Applications of Queue

- ◆ Queue is used in CPU scheduling where processes are managed using ready and waiting queues. The first process to enter is given CPU time first, following scheduling algorithms like FCFS and Round Robin.
- ◆ In print spooling, when multiple print jobs are sent to a shared printer, they are stored in a queue. The printer processes them one by one in the order they arrive.
- ◆ Web servers use queues to handle multiple client requests. Each request is placed in a queue and served based on arrival time, ensuring fair and sequential processing.
- ◆ In video streaming and I/O buffering, queues temporarily store data so that it can be processed smoothly. For example, video data is buffered in a queue to allow uninterrupted playback.
- ◆ In graph algorithms like Breadth-First Search (BFS), a queue is used to keep track of nodes to be explored next, allowing level-by-level traversal of the graph.
- ◆ Queues are used in real-world service simulations such as bank counters, ticket booking systems, and hospital registrations, where customers are served one after another.

2.4.4 Implementation of queue

A queue can be implemented in C using both array and linked list. In an array-based implementation, a fixed-size array is used along with two variables `front` and `rear` to track the positions for insertion and deletion. Elements are added at the rear and removed from the front, following the FIFO principle. However, this approach has a limitation of fixed size and inefficient use of space after several deletions. In contrast, a **linked list-based implementation** uses dynamic memory allocation, where each element (node) contains data and a pointer to the next node. The `front` points to the first node, and `rear` points to the last. Insertion happens at the rear, and deletion happens at the front, just like in the array-based method. This approach is more flexible as it allows the queue to grow and shrink as needed, avoiding the problem of fixed size and wasted space.

2.4.5 Types of Queue

The queue is a linear data structure used to represent a linear list. It allows the insertion of an element to be done at one end and deletion of an element to be performed at the other end. In a queue, the first element inserted is the first one to be deleted or removed. Therefore, a queue follows the FIFO (First In, First Out) structure. The queue data structures are classified into four categories. They are;

1. Simple Queue
2. Circular Queue
3. Double Ended Queue (D - Queue)
4. Priority Queue

2.4.5.1 Simple queue

In a simple queue, the insertion occurs at the rear end of the queue, and deletion occurs at the front end of the queue. In a simple queue, adding an element in a queue is called enqueue operation, and the process of removing an element from a queue is called dequeue operation. A simple queue follows the FIFO (First in, First out) rule. The best example of a simple queue process is in the checkout line at the supermarket cash counter, where the first person in the line is usually the first to be checked out. Figure 2.4.7 is the graphical representation of a simple queue.

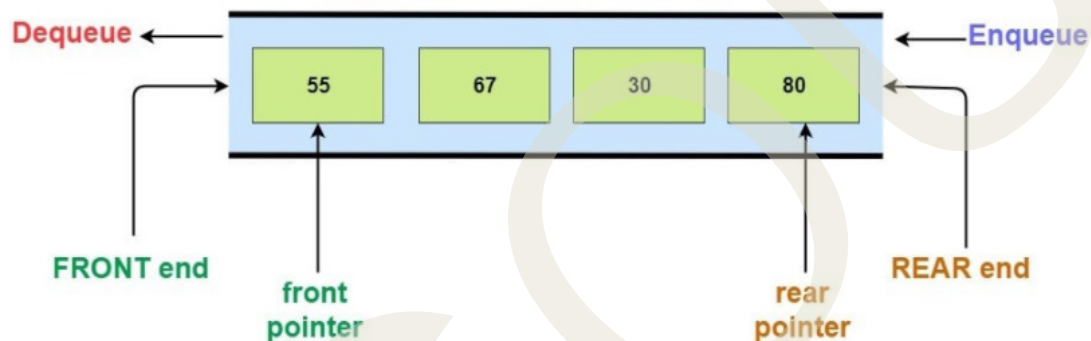


Fig 2.4.7 Simple Queue

The simple queue contains two pointers, front pointer and rear pointer. The front pointer includes the location of the front element of the queue, and the rear pointer consists of the location of the rear element of the queue. The algorithm for insertion and deletion of data in a simple queue are given below:

The condition $FRONT = -1$ will indicate Q is Empty.

The condition $REAR \geq N-1$ will indicate Q is Full.

Where, N represents the maximum capacity or the total number of elements that the queue can hold

The simple queue insertion process (enqueue algorithm) Q, N, F, R and Item are the variables used in this algorithm creation. Here F and R are front and rear elements of the queue. The queue 'Q' contains N data elements. The 'item' is the new item in the simple queue list at the rear end of the queue.

QINSERT (Q,N,F,R,Item)

Initially $F = R = -1$

Step1: [Check for Overflow Condition]

If $R \geq N-1$ then:

write "Overflow"

return

Step2: [Increment REAR Pointer]

If $F == -1$ then : [Queue initially empty]

set $F = 0$ and $R = 0$

else :

set $R = R + 1$

[End of If Structure]

Step 3: [Insert Item]

set $Q[R] = \text{Item}$

Step 4: Return

The simple queue deletion process (dequeue algorithm) Q, N, F , and R are used in this algorithm creation. Here F and R are front and Rear elements of the queue. The queue Q consists of N elements. This procedure deletes an element from the front end of the queue.

QDELETE (Q, N, F, R)

Initially $F = R = -1$

Step 1: [Check for Underflow Condition]

If $F == -1$ then: Design and Explain Prim's Algorithm for Finding the Minimum Spanning Tree (MST) of a Graph

write "Underflow"

return (0)

Step 2: set $Y = Q[F]$ and

$[F] = \text{NULL}$

Step 3: [Increment Front Pointer]

If $F == R$ then : [Queue has only one element left]

set $F = R = -1$

else :

set $F = F + 1$

[End of If Structure]

Step 4: Return (Y)

An example of a simple queue in the real world is a single-lane one-way road, where

vehicles enter and exit in the order they arrive. More real-world examples can be seen as queues at the ticket windows and bus stops. If the simple queue is full, that is $R == N-1$, and if you delete elements from the simple queue's front end, then there are some free spaces available at the front end in the queue. This free space cannot be used for inserting new elements. This is the one drawback of a simple queue.

2.4.5.2 Circular Queue

A circular queue is a queue that is implemented in a circular form. It is also a linear data structure, which follows the principle of FIFO (First In First Out). Instead of ending at the last position, the queue starts again from the first position, making it behave like a circular data structure. Figure 2.4.8 is the Graphical representation of a circular queue.

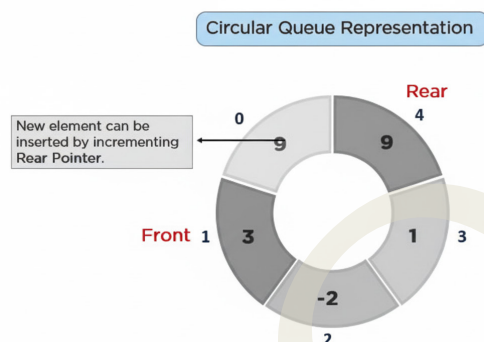


Fig 2.4.8 Circular Queue

The circular queue structure overcomes the problem of unutilized space in a simple linear queue implemented as an array. In the array implementation, it's possible for the queue to be reported as full even when there are empty slots available in the queue. This is because, at the time, the rear part of the queue has reached the end of the array. When the queue gets full, there is a possibility of the vacant element at the beginning of the array if some of the elements are deleted.

Figure 2.4.8 represents a data organizational graphical image of a circular queue, the queue having six data elements, its rear value is 50, and the front value is 23. A circular queue is similar to a linear queue because it is based on the FIFO (First in First Out) principle except that the last position is connected to the first position. But all circular queue data structures are not linear.

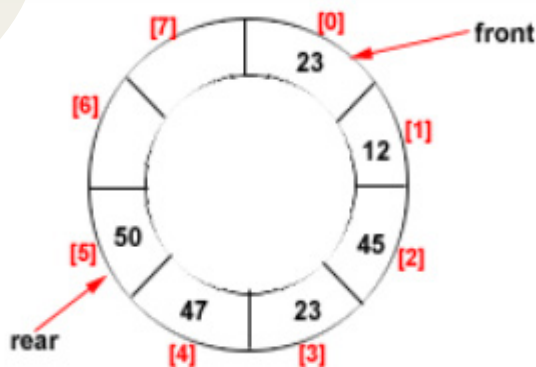


Fig 2.4.9 Circular Queue having Rear = 5 and Front = 0

The following are the operations that can be performed on a circular queue. In the circular queue, there are two pointers, front and rear. The front pointer is used to get the front(first) element from the circular queue, and the rear pointer is used to get the rear (last) element from the circular queue. Thus, the queue has insertion and deletion operations; they are enqueue and dequeue.

The enqueue operation is used to insert the new value into the circular queue. The new element is always inserted from the rear end. This operator works as follows:

Step1: Check if the queue is full

Step2: For inserting the first element, set the value of the front to 0

Step3: Circularly increase the rear index value by 1

Step4: Add the new element in the position pointed to by the rear pointer.

The dequeue operation is used to delete an element from the circular queue. The deletion in a queue always takes place from the front end. This operator works as follows;

Step1: Check if the queue is empty

Step2: Return the value pointed by the front pointer

Step3: Circularly increase the front index value by 1

Step4: For the last element, reset the values of the front pointer and a rear pointer to -1

Enqueue and dequeue operations are based on the queue data insertion and deletion algorithms in the circular queue.

The two insertion and deletion algorithms are given below:

1) The Circular Queue Data Insertion Algorithm

In this data insertion process, CQueue, Rear, Front, N and Item are the variables used in this algorithm creation. First, insert data elements into the queue; CQueue is a circular queue where to store data. The Rear represents the location in which the data element is to be inserted. The front represents the location from which the data element is to be removed. Finally, N is the maximum size of the queue, and the item is the new item in the circular queue list.

Insert Circular Q(CQueue, Rear, Front, N, Item)

Initially Rear = 0 and Front = 0.

Step1: If Front = 0 and Rear = 0 then Set Front = 1 and go to step 4.

Step2: If Front = 1 and Rear = N or Front = Rear + 1

then Print: “Circular Queue Overflow” and Return.

Step3: If Rear = N then Set Rear = 1 and go to step 5.

Step4: Set Rear = Rear + 1

Step5: Set CQueue [Rear] = Item.

Step6: Return

2) The Circular Queue Data Deletion Algorithm

After inserting all the data elements into the circular queue, the queue executes a delete operation. In this data deletion process, CQueue, Rear, Front, N and Item are the variables used in the algorithm. The CQueue is a circular queue where to store data. The rear represents the location in which the data element is to be inserted, and the front represents the location from which the data element is to be removed. The front element in the queue is assigned to the item.

Delete Circular Q(CQueue, Front, Rear, Item)

Initially, Front = 1.

Step1: If Front = 0 then

Print: "Circular Queue Underflow" and Return.

Step2: Set Item = CQueue [Front]

Step3: If Front = N then Set Front = 1 and Return.

Step4: If Front = Rear then Set Front = 0 and Rear = 0 and Return.

Step5: Set Front = Front + 1.

Step6: Return.

The following graphical representation is the data insertion and deletion process in a circular queue data structure. For example, the below given circular queue is having five data locations, i.e. $N=5$.

1. Initially, Rear = 0, Front = 0.

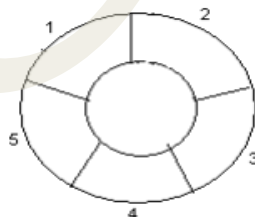


Fig 2.4.10: illustration of step1

2. Insert 10, Rear = 1, Front = 1.

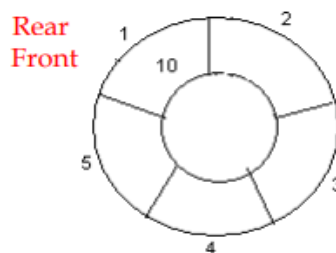


Fig 2.4.11: illustration of step2

3. Insert 50, Rear = 2, Front = 1.

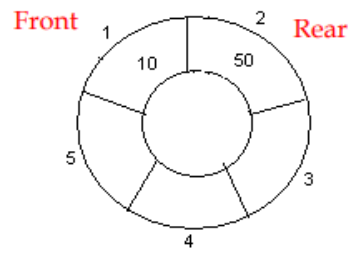


Fig 2.4.12 Illustration of step3

4. Insert 20, Rear = 3, Front = 1.

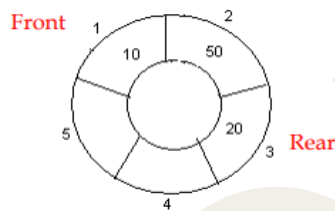


Fig 2.4.13 Illustration of step4

5. Insert 70, Rear = 4, Front = 1.

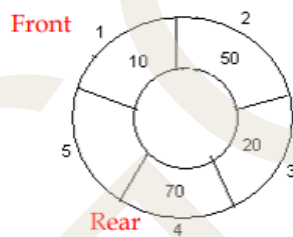


Fig 2.4.14 Illustration of step5

6. Delete Front(10), Rear = 4, Front = 2.

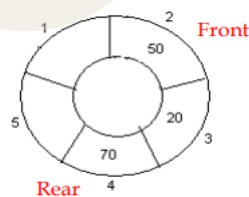


Fig 2.4.15 Illustration of step6

7. Insert 100, Rear = 5, Front = 2.

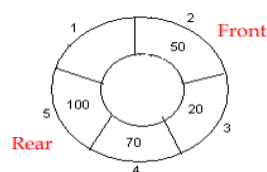


Fig 2.4.16 Illustration of step7

8. Insert 40, Rear = 1, Front = 2.

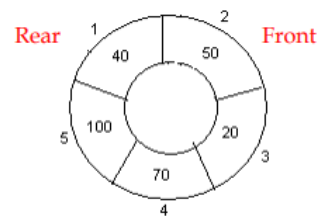


Fig 2.4.17: illustration of step8

9. Insert 40, Rear = 1, Front = 2.

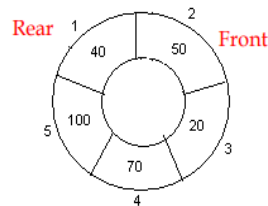


Fig 2.4.18 Illustration of step9

As Front = Rear + 1, so Queue overflow.

10. Delete Front (50), Rear = 1, Front = 3

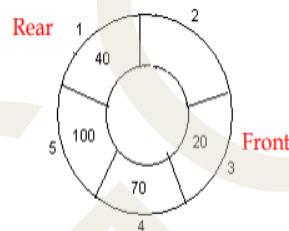


Fig 2.4.19: illustration of step10

11. Delete Front(20), Rear = 1, Front = 4.

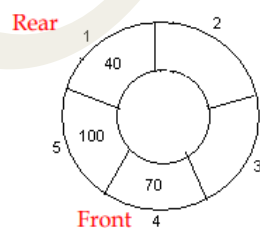


Fig 2.4.20: illustration of step11

12. Delete Front(70), Rear = 1, Front = 5.

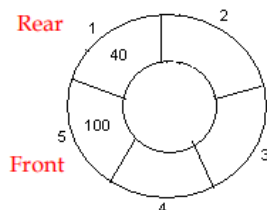


Fig 2.4.21 Illustration of step12

A traffic light sequence on a large roundabout is an example of a circular queue. It changes the signals circularly in a sequence with an equal interval of time. Some other real-time examples are print spooler of the operating system, bottle-capping systems in cold drink factories, a routine of human life and days in a week.

2.4.5.3 Double-Ended Queue

Figure 2.4.21 shows the double-ended queue (Deque) graphical image. It is the third type of data structure in the queue. A double-ended queue is a linear list in which elements can be added or removed at either end but not in the middle. The double-ended queue is also called D-queue or DE-queue or deque, depending upon the operations. The double-ended queue operations are categorized into two types: input restricted deque and output limited deque.

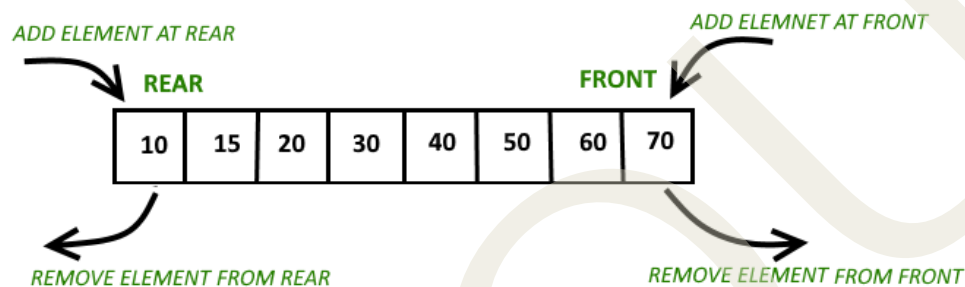


Fig 2.4.22 Double Ended Queue

1. Input Restricted Deque

As the name suggests, in an input-restricted queue, insertion operations are limited to one end, while deletions can occur from both ends. Thus, data can be inserted at one end (say REAR end) only, but data can be deleted from both ends as shown in figure 1.4.22.



Fig 2.4.23 The Input Restricted Deque

2. Output Restricted Deque

An output restricted Deque is a type of double ended queue where deletions take place at one end only say (FRONT end), but allows insertions at both ends. Figure 1.4.23 represents an output restricted deque.



Fig 2.4.24 The Output Restricted Deque

One of the real-world applications of the double-ended queue is storing a web browser's history. Recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front. Another common application of the deque is storing a software application's list of undo operations.

2.4.5.4 Priority queue

The priority queue is the fourth type of data structure. It is an extension of the queue data structure. This data structure is a particular type in which each element has been assigned a value, called priority of the element, and an element can be inserted or deleted not only at the ends but at any position on the queue. Figure 1.4.24 shows a priority queue.



Fig 2.4.25 Priority Queue

With this structure, an element x of priority p_i may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority, that is instead of adding it after the REAR it may be inserted at an intermediate position dictated by its priority value. A priority queue does not strictly follow the first-in-first-out (FIFO) principle. Which is the basic principle of queue data structure. Instead, the priority queue operates and arranges the elements so that 'high-priority elements are served before the 'low-priority elements.

2.4.6 Comparison of different types of queues

Table 2.4.1 Comparison of types of queues

Type of queue	Structure & Operation	Insertion	Deletion	Features
Simple Queue	Linear structure where elements are inserted at the rear and removed from the front.	Rear	Front	Follows FIFO (First-In-First-Out) principle.
Circular Queue	Last position is connected back to the first position to make a circle.	Rear	Front	Avoids wastage of space in fixed-size arrays.

Double Ended Queue	Also called Deque ; insertion and deletion are possible at both ends.	Both front and rear	Both front and rear	More flexible; can act as queue or stack depending on usage.
Priority Queue	Elements are arranged based on priority , not just arrival time.	According to priority	Highest (or lowest) priority	Not strictly FIFO; highest priority is served first.

Recap

- ◆ A queue is a linear data structure that follows the First in First Out (FIFO) rule.
- ◆ Elements are added at the rear and removed from the front.
- ◆ The first element inserted is the first one to be removed.
- ◆ A common example is people waiting in line at a ticket counter.
- ◆ The enqueue operation adds an element at the rear.
- ◆ The dequeue operation removes an element from the front.
- ◆ The peek() function shows the front element without removing it.
- ◆ The isfull() function checks if the queue is full.
- ◆ The isempty() function checks if the queue is empty.
- ◆ In enqueue, if the queue is not full, the element is inserted and the rear is increased.
- ◆ In dequeue, if the queue is not empty, the front element is removed and the front is increased.
- ◆ Queues are used in CPU scheduling to manage running processes.
- ◆ In print spooling, print jobs wait in a queue to be printed.
- ◆ Web servers use queues to handle requests one after another.
- ◆ In video streaming, queues help buffer data for smooth playback.
- ◆ Breadth-First Search (BFS) in graphs uses a queue to visit nodes level by level.
- ◆ A simple queue allows insertion at the rear and deletion at the front.
- ◆ A circular queue connects the end to the beginning to reuse space.

- ◆ A double-ended queue (deque) allows insertion and deletion at both ends.
- ◆ A priority queue removes elements based on priority, not just the order of arrival.
- ◆ In a priority queue, elements with higher priority are removed first.
- ◆ A priority queue does not strictly follow the FIFO rule.

Objective Type Questions

1. Name the linear data structure that uses the First-In-First-Out method
2. What is the term used for inserting an element into a queue?
3. State the operation used to remove an element from a queue
4. What do you call a queue that connects the last position to the first?
5. Which function is used to check if the queue has no elements?
6. Which function returns the front element without removing it?
7. Name the queue where insertion and deletion can occur at both ends
8. What is the condition called when insertion is attempted in a full queue?
9. Which type of deque restricts insertion to one end only?
10. Which deque allows insertion at both ends but deletion at only one?
11. Name the queue that organizes elements based on priority rather than order of arrival.
12. What error occurs when trying to remove an item from an empty queue?
13. Which pointer is updated after an insertion in a queue?
14. Which pointer moves forward after an element is removed from the queue?
15. Which data structure is used in Breadth-First Search for storing nodes?
16. What term refers to the total number of elements a queue can hold?
17. Which queue structure serves elements based on priority rather than position?

Answers to Objective Type Questions

1. Queue
2. Enqueue
3. Dequeue
4. Circular
5. isempty()
6. peek()
7. Deque
8. Overflow
9. Input-restricted
10. Output-restricted
11. Priority
12. Underflow
13. Rear
14. Front
15. Queue
16. Capacity
17. Priority

Assignments

1. Explain the working of a queue data structure with the help of a real-life example?
2. Describe its basic operations such as enqueue, dequeue, peek, isfull, and isempty with proper algorithmic steps.
3. Discuss the implementation of a simple queue using a one-dimensional array?

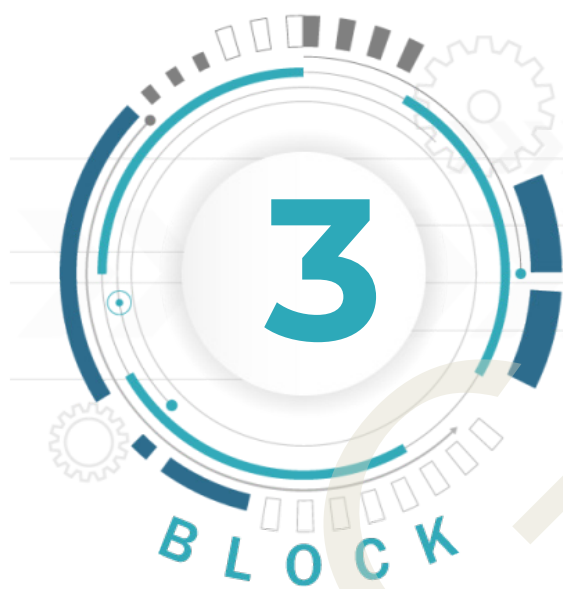
4. What is a circular queue? How does it overcome the drawbacks of a simple queue?
5. Write a detailed note on the applications of queue data structure in real-world and computer science domains.
6. Write a short note on different types of queues?

Reference

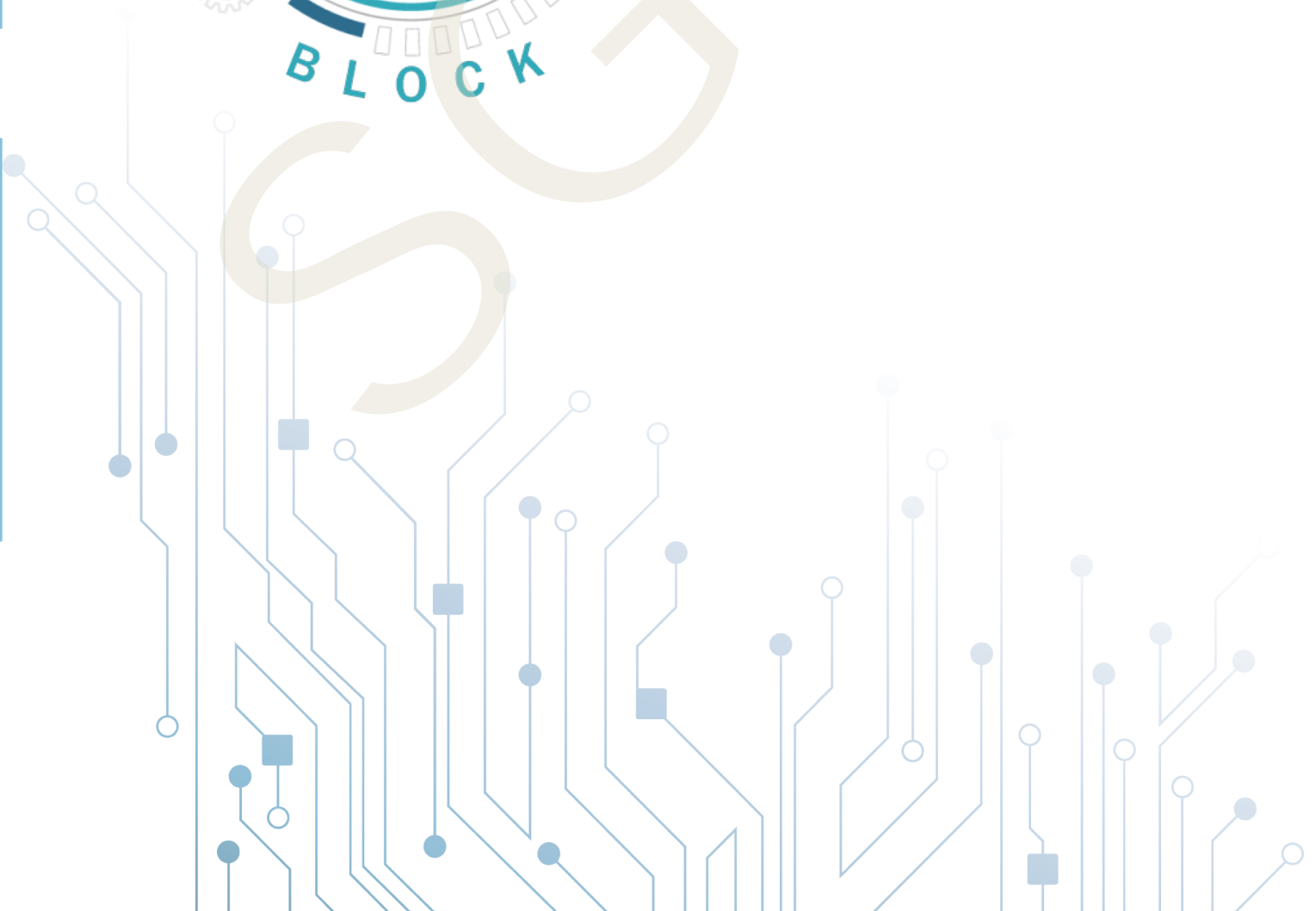
1. Rao, G. A. V. (2021). *Data structures and algorithms using C and C++*. PHI Learning.
2. Narayan, S. (2021). *Data structures and algorithms made easy: Data structure and algorithmic puzzles* (5th ed.). CareerMonk Publications.
3. Tiwari, K. (2021). *Data structures, algorithms and programming techniques in C*. BPB Publications.
4. Thareja, R. (2019). *Data structures using C* (2nd ed.). Oxford University Press.
5. Lipschutz, S. (2014). *Data Structures (Schaum's Outlines Series)*. McGraw-Hill Education.

Suggested Reading

1. Narayan, S. (2021). *Data structures and algorithms made easy: Data structure and algorithmic puzzles* (5th ed.). Career Monk Publications.
2. Lipschutz, S. (2014). *Data structures* (Schaum's outlines series, Revised ed.). McGraw-Hill Education.
3. Kruse, R. L., Leung, B. P., & Tondo, C. L. (1999). *Data structures and program design in C* (2nd ed.). Pearson Education.
4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.



Linear Data Structures



Unit 1

Linked List Basics

Learning Outcomes

After completing this section, learners will be able to:

- ◆ make aware of the concept of Linked List
- ◆ familiarise the representation and implementation of Linked List
- ◆ describe Self-referential structures.
- ◆ explain the traversal on a Linked list.

Prerequisites

How can data be organized in a way that allows for efficient processing and multiple operations? How can it be stored so that retrieving and working with it becomes easier? This is where the concept of data structures comes into play.

Data structures are methods of organizing and storing data to enable efficient access and manipulation. By structuring data properly, tasks can be completed more effectively and future operations become simpler. For instance, if we need to maintain data in a sequential order, we might use a list. A list is an example of an abstract data type, just like arrays, stacks, and queues, which are also used to handle data in specific ways.

The images below display several items organized into groups, with each group arranged according to a specific strategy.



Fig 3.1.1 Queue

Fig.3.1.1. **Queue** shows a new person will join at the end of the queue.





Fig 3.1.2 Stack of Books

Fig. 3.1.2 illustrates that a new book can be added on top of the existing stack of books.

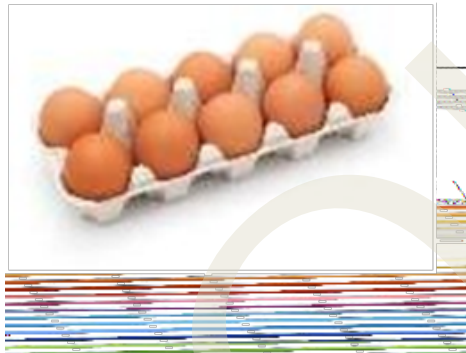


Fig 3.1.3 Heap of stones

Fig. 3.1.3 illustrates a heap of stones that appears to be piled up randomly, without following any particular arrangement.



Fig.3.1.4 Egg Rack

Fig. 3.1.4 illustrates a tray holding eggs, where each egg can be placed in any slot of the rack. This visual represents how a data structure functions, similar to organizing a collection of items where each element has a specific position.

In this unit, we will explore a new type of data structure known as the Linked List.

Imagine a movie theatre where seats are arranged in a fixed sequence like an array. Tickets are issued strictly in the order of seat numbers, meaning they must be allocated sequentially. Now, if some people cancel their tickets and others want to book new ones, the system may not allow new bookings - even if some seats are vacant - because the available seats are not in a continuous sequence. This leads to an inefficient and unfair situation.

A similar limitation exists with arrays: even if there are multiple empty memory slots,

data can only be stored if those slots are adjacent. This is where linked lists become useful.

Unlike arrays, a linked list connects data elements using pointers or links, not by storing them in consecutive memory locations.

Key words

Node, Pointer, Self-referential structure, Traversal, Dynamic Memory Allocation

Discussion

3.1.1 Linked List

- ◆ Let us now explore the various types of linked lists and how information is stored within them.

Imagine we want to store student names in memory. There are two primary ways to manage this list in memory:

- Using an Array
 - Using a Linked List
- ◆ You are already familiar with arrays. An array is a fixed-size data structure where elements are stored in consecutive memory locations, and the number of items is limited.

In contrast, a linked list is a dynamic structure with no fixed size. Unlike arrays, the elements of a linked list are not stored in sequence but are scattered in memory, with each element connected using pointers. These pointers store the address of the next data item in the sequence. A linked list expands as new data is added and contracts when data is removed. This unit will cover the different types of linked lists in detail.

- ◆ Some of these types include Singly Linked List, Doubly Linked List, and Circular Linked List.

The concept of a singly Linked List

A singly linked list is a fundamental type of linked list where traversal is restricted to the forward direction only, meaning we can move ahead but not backward. In contrast, a doubly linked list allows movement in both forward and backward directions.

A singly linked list, often simply called a linked list, is composed of nodes. Each node has two components:

- ◆ Data
- ◆ Link

What do we mean by a Node? A node is represented as shown in Fig. 3.1.5.

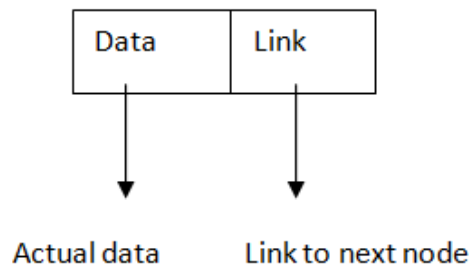


Fig. 3.1.5 Node representation

The data section holds the actual information, while the link section stores the address of the next node in the list. Therefore, a linked list is made up of nodes, where each node includes data and a link, which points to the following node in the sequence. The initial node in a linked list is referred to as the Head.

3.1.2 Representation of Linked List

Imagine we want to keep a list of student names in memory - Athul, Hima, Manu, and Rihan. For this purpose, we must create four separate nodes.



Fig 3.1.6 Four node examples of student names

The first node holds the data 'Hima' in its data section, while the second node stores 'Manu', and so on. These nodes don't need to be stored in consecutive memory locations. For illustration, node addresses are labeled as 100, 102, 104, and 106. Being a linked list, the memory addresses are not in a sequence. The first node is located at address 100, and the last one at 106.

The nodes are distributed across memory. To navigate through the list, each node must store the address of the next node in its link section. The link field of the first node holds the address of the second, the second points to the third, and so forth.

The final node's link section contains NULL, which signals the end of the list. This means we can move from the first to the second node using the address stored in the link part, and continue this process until we encounter NULL.

A NULL value in a link field indicates the termination of the list.

But how is the first node located? This is done using a pointer called START, which stores the address of the first node in the linked list.

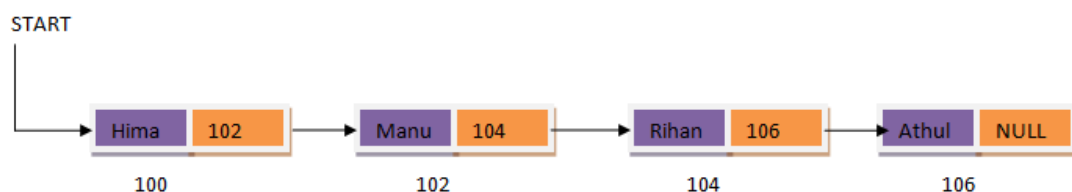


Fig.3.1.7 Four node example of Linked List

3.1.3 Implementation of LinkedList

How is a node created for a linked list in C? A linked list consists of multiple nodes, each having two parts: one holds the data, and the other stores a pointer to the next node in the sequence.

To implement this in C, we make use of a structure. As we know, a structure is a user-defined data type. One of its members can be a pointer, and it can point to the same type of structure. This is known as a self-referential structure.

How is a Node defined in C?

We use a self-referential structure to define a node in a linked list.

A self-referential structure includes a pointer that points to another instance of the same structure type.

Example:

```
struct sample {  
    int a; char b;  
    struct sample *self;  
};
```

In this example, 'sample' is a structure that contains a pointer named 'self' which refers to another 'sample' structure. Therefore, it is called a self-referential structure.

Node

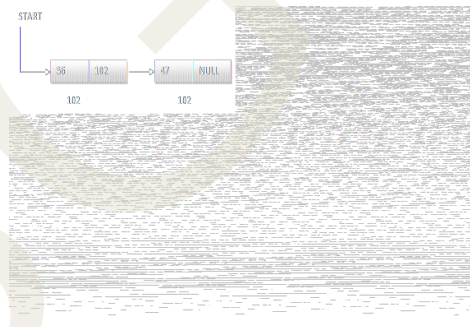


Fig 3.1.8 Self-Referential Structures

As we have learned, a node combines two distinct types of data—it includes both a data part and a link part. To merge different data types into a single unit, we use a structure. The nodes of a linked list can be created using a structure as shown below:

```
struct node {  
    int data1;  
    char data2;
```

```

struct node *link;
}

```

This structure, **struct node**, contains three components: the first is an integer, the second is a character. These data fields can vary in type and quantity—such as char, float, or any other type. The third component, **link**, is a pointer to another node of the same structure type. But what exactly is **struct node *link**?

The link serves as a pointer to another node, which, as discussed, is itself a structure. Therefore, the link must point to a **struct node**, making it a self-referential structure, as illustrated in Figure 3.1.8.

3.1.3.1 Creation of a Linked list - Algorithm

Building a linked list involves repeatedly inserting new nodes at the end of the list. The steps involved in the process are as follows:

Step 1: Allocate memory for a new node and obtain its address.

Step 2: Insert the data into the data field and assign NULL to the link field of the node.

Step 3: If it is the first node, store its address in the START pointer.

Step 4: If it is not the first, update the link field of the previous node with the new node's address.

Step 5: Continue repeating steps 1 to 4 until the user chooses to stop.

3.1.3.2 Linked list creation steps

1. Linked list is empty

START

2. The first node is created with address 100. Data and links are filled

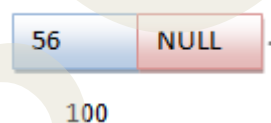


Fig 3.1.9(a) Creation of a Linked list

3. The address of the first node, here=100, is stored in the START

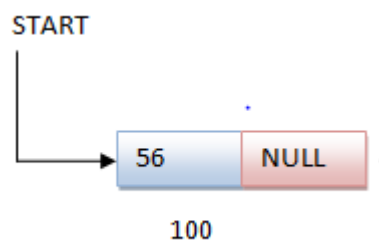
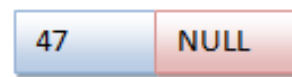


Fig 3.1.9(b) Creation of a Linked list

4. The second node is created with address 102. Data and link are filled



102

Fig 3.1.9(c) Creation of a Linked list

5. The address of the Second node is stored in the link part of the first node.

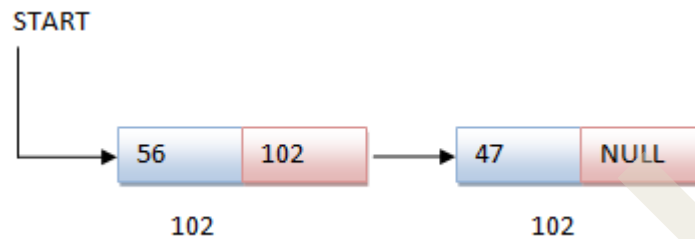
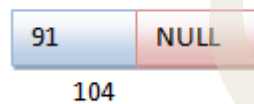


Fig 3.1.9(d) Creation of a Linked list

6. The third node is created with address 104, and Data and Link are filled.



104

Fig 3.1.9(e) Creation of a Linked list

7. The address of the **third node** is stored in the **link part of the second node**.

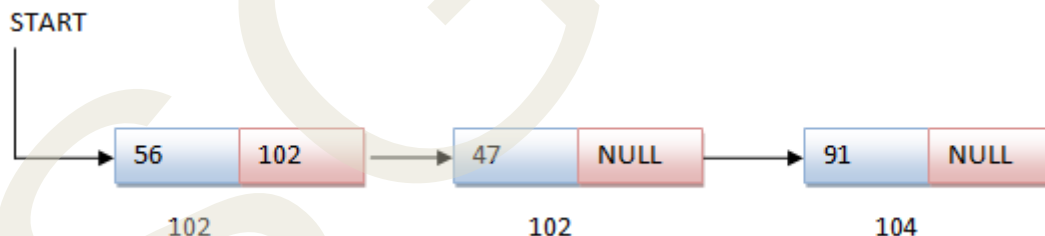


Fig 3.1.9(f) Creation of a Linked list

Stacks and queues can also be implemented using a linked list, resulting in dynamic versions of these structures. In this section, we focus on creating a singly linked list. The next unit will cover how to insert and delete nodes within the singly linked list.

Creating a node in C

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
struct node {
```

```

int data;          // 'data' holds the value stored in the node

struct node *link; // 'link' is a pointer to the next node
};

int main() {

    struct node *start = NULL; // 'start' points to the first node; initialized to NULL

    start = (struct node *) malloc(sizeof(struct node)); // allocate memory for one node
    using malloc; assign its address to 'start'

    start->data = 56; // assign the value 56 to the data part using the pointer with '->'

    start->link = NULL; // the link part is set to NULL indicating there's no next node

    printf("%d", start->data); // print the data stored in the first node

    return 0;
}

```

3.1.4 Traversal on Linked List

Traversing a singly linked list involves accessing each node one by one until the final node is reached. In such a list, traversal starts at the first node. The 'Start' pointer holds the address of this node, allowing us to access its data using the arrow operator. The link part of this first node provides the address of the next node. Using that address, we can access both the data and the link of the second node. This process continues node by node until we encounter a NULL pointer in the link section, indicating the end.

To make this clearer, let's consider a simple example. Suppose we already have a linked list available, and our goal is to traverse it. While doing so, we also want to count how many nodes exist in the list. The diagram below outlines the steps necessary to perform the traversal operation. It is assumed that Temp is a temporary pointer of type 'node', and Val is a variable used to store the data retrieved from each node.

(1) Linked List having three nodes.

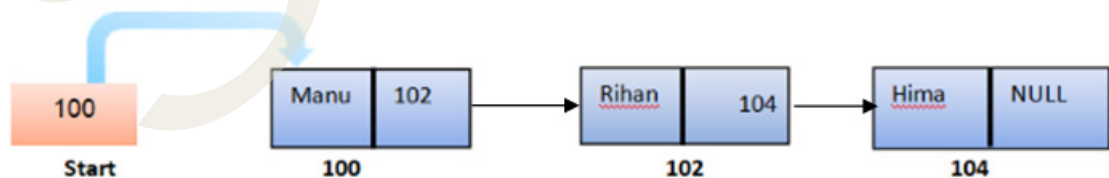


Fig 3.1.10(a) Showing traversal operation in a linked list

(2) The content of the start is copied into Temp (Because if we change the value of the start, the address of the first node will be lost when traversing). Now, Temp can point to the first node.

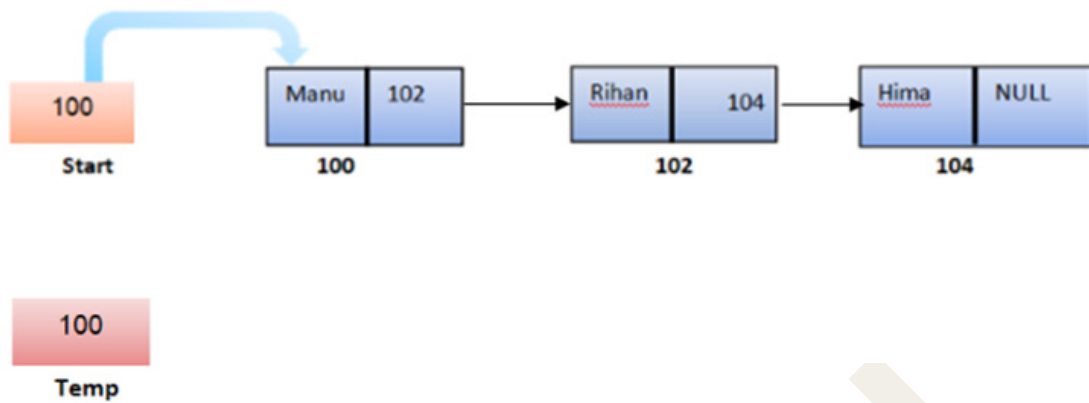


Fig 3.1.10(b) Showing traversal operation in a linked list

(3) The first node is accessed using Temp, and Data is retrieved to Val. After that, the Temp is updated by the address of the second node.

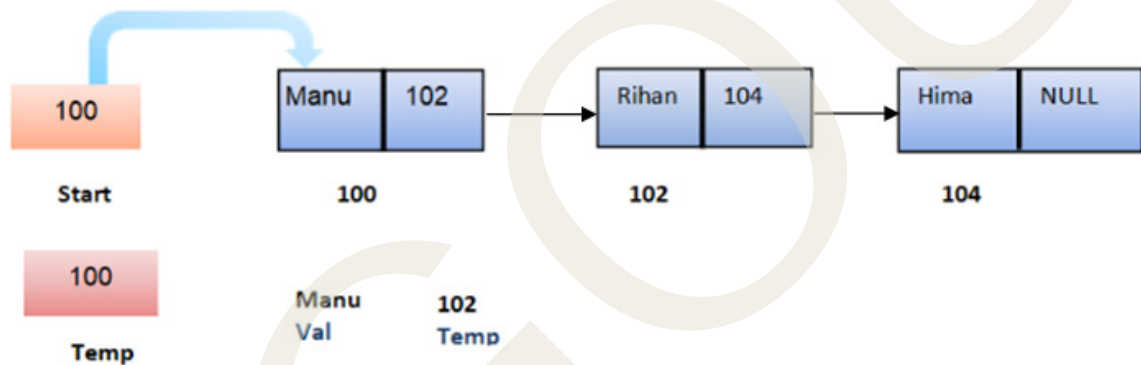


Fig 3.1.10(c) Showing traversal operation in a linked list

(4) The second node is accessed using Temp, and data is retrieved. After that, the Temp is updated by the address of the third node.

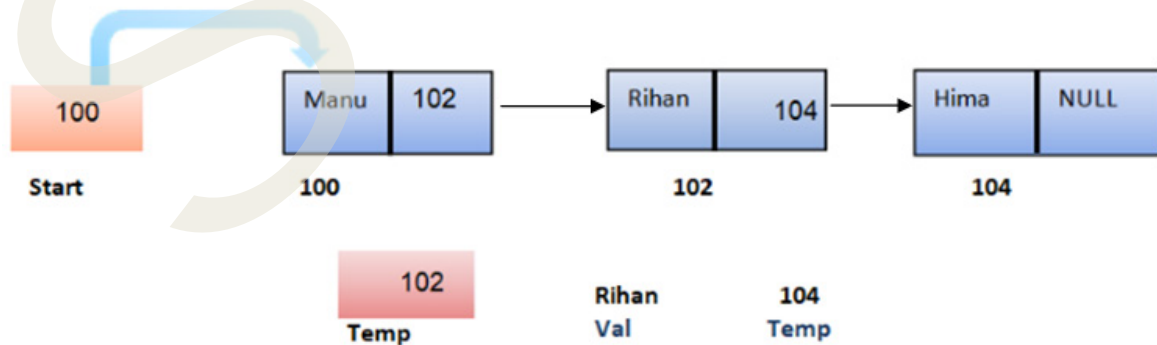


Fig 3.1.10(d) Showing traversal operation in a linked list

(5) Third node is accessed using value in Temp, and data is retrieved to Val. After that, the link part of the third node updates that Temp. Since it is NULL, traversal ends here.

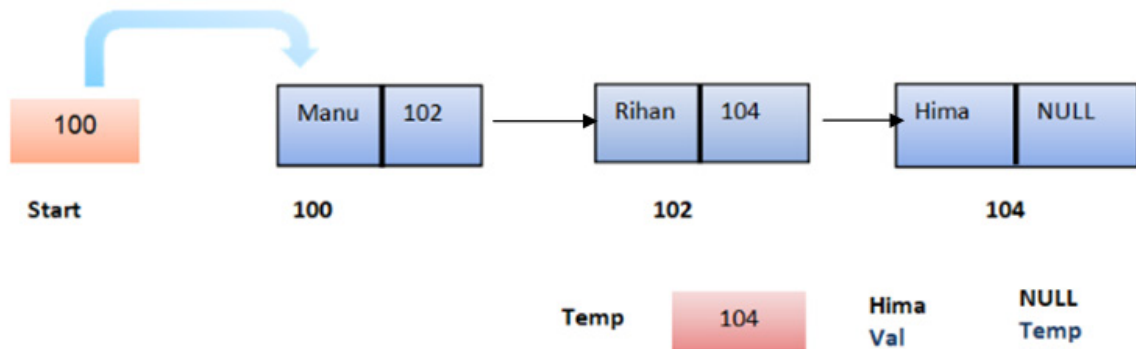


Fig 3.1.10(e) Showing traversal operation in a linked list

3.1.4.1 Traversal operation in linked list - Algorithm

- Step 1: Retrieve the address of the first node from Start and assign it to Temp.
- Step 2: Access the data of the current node using Temp and store it in Val.
- Step 3: Get the address from the link part of the current node and update Temp with it.
- Step 4: Repeat step 2 until Temp contains NULL; otherwise, terminate the process.

The above steps describe how to traverse a linked list when Start is not NULL. In such cases, reaching the last node is essential to attach the address of a new node in its link field. The traversal begins from the first node, with its address stored in a temporary pointer (Temp as shown in the figure). The pointer is updated by assigning it the value from the current node's link field. This continues until Temp points to a node whose link is NULL.

Recap

- ◆ Linked lists are dynamic data structures used to store elements connected by pointers.
- ◆ Arrays have fixed sizes and store elements in contiguous memory locations.
- ◆ Linked lists do not require contiguous memory and can grow or shrink as needed.
- ◆ Each element in a linked list is called a node.
- ◆ A node consists of two parts: data and a link (pointer to the next node).
- ◆ The first node of the linked list is referred to as the Head or START.
- ◆ Singly linked lists allow traversal in only one direction—from start to end.
- ◆ Doubly linked lists support traversal in both forward and backward directions.
- ◆ Circular linked lists connect the last node back to the first, forming a loop.

- ◆ Linked lists are useful for dynamic memory allocation and frequent insertions or deletions.
- ◆ Node structures in C are created using self-referential structures.
- ◆ A self-referential structure includes a pointer to another structure of the same type.
- ◆ The structure for a node typically includes data fields and a pointer to the next node.
- ◆ Creating a linked list involves allocating memory, assigning data, and updating links.
- ◆ The START pointer holds the address of the first node in the linked list.
- ◆ The link part of each node stores the address of the next node.
- ◆ The last node's link field contains NULL, indicating the end of the list.
- ◆ Traversing a linked list means accessing each node sequentially until the end.
- ◆ A temporary pointer (e.g., Temp) is used to traverse without losing the START address.
- ◆ Linked list traversal is useful for operations like searching, counting nodes, or displaying data.

Objective Type Questions

1. What data structure allows dynamic memory allocation unlike arrays?
2. What is the fixed-size data structure where elements are stored in contiguous memory?
3. What component of a node stores the address of the next node?
4. What is the first node of a linked list commonly called?
5. Which special pointer in C is used to dynamically allocate memory for a node?
6. What keyword is used in C to define a structure?
7. What kind of structure refers to itself in one of its members?
8. What type of linked list allows only forward traversal?

9. What is the value of the link field of the last node in a linked list?
10. Which field of a node holds the actual information?
11. What does the START pointer hold in a linked list?
12. Which type of linked list allows traversal in both directions?
13. What operator is used to access structure members through a pointer?
14. Which type of data structure is implemented using nodes and links?
15. Which function is used in C to allocate memory dynamically?
16. What does a NULL link indicate in a linked list?
17. What is the name of the temporary pointer used during traversal?
18. What operation involves visiting each node of a linked list?
19. In a singly linked list, how many links are present in each node?
20. Which part of the node helps in navigating from one node to another?

Answers to Objective Type Questions

1. LinkedList
2. Array
3. Link
4. Head
5. Malloc
6. Struct
7. Self-referential
8. Singly
9. NULL
10. Data

11. Address
12. Doubly
13. Arrow
14. LinkedList
15. Malloc
16. End
17. Temp
18. Traversal
19. One
20. Pointer

Assignments

1. Explain the differences between arrays and linked lists in terms of memory allocation, data access, and size flexibility. Provide suitable examples.
2. Define a singly linked list. Describe how a node is represented and linked to other nodes using pointers in C.
3. Write the algorithm for creating a singly linked list. Explain each step with a suitable diagram.
4. What is a self-referential structure in C? How is it useful in implementing linked lists? Provide a code example.
5. With the help of diagrams, explain how nodes are connected in a singly linked list. What role does the START pointer play in this connection?
6. Discuss the process of traversing a singly linked list. Write the algorithm and explain with an example how node data is accessed during traversal.
7. Illustrate the memory representation of a singly linked list storing student names. Explain how nodes are distributed and accessed.
8. Explain how stacks and queues can be implemented using linked lists. What are the advantages of using linked lists over arrays in such implementations?

Suggested Reading

1. “Data Structure using C” by A K Sharma.
2. “Data Structures and Program Design in C” by Kruse Robert L.
3. “Data Structures and Algorithms Analysis in C” by Mark Allen Weiss.
4. “Data Structures and Algorithms” by Alfred V Aho and Jeffrey D Ullman

Unit 2

Operations on Linked List

Learning Outcomes

After completing this section, learners will be able to:

- ◆ define a linked list and its basic components
- ◆ list the different types of linked list operations
- ◆ identify the steps required to insert or delete a node
- ◆ recall the differences between arrays and linked lists

Prerequisites

Imagine you are building a music playlist app where users can freely add, remove, or rearrange songs. Initially, using an array to store the playlist might seem like a good choice. However, as users start inserting songs in the middle, deleting the first track, or expanding the playlist beyond its original size, arrays become less efficient because of their fixed size and the need to shift elements around. This situation reveals the importance of a more flexible data structure called a linked list. Unlike arrays, linked lists allocate memory dynamically and make inserting and deleting elements at any position much easier. In this lesson, learners will explore linked lists, learn how they are structured and function, and practice key operations such as insertion, deletion, searching, and sorting. By the conclusion, students will understand when linked lists are a better choice than arrays and gain essential skills for developing efficient algorithms and effective memory management in their programs.

Key words

Dynamic Memory Allocation, Insertion, Deletion, Traversal, Sorting



Discussion

Operations on a linked list involves modifying its nodes to carry out different tasks. The main operations performed on a linked list include

1. Insertion in a linked list
2. Deletion from a linked list
3. Searching in a linked list
4. Sorting of elements in linked list

3.2.1 Insertion in a linked list

Inserting an element into a linked list involves adding a new node at a specific location within the list. This process requires three main steps:

1. Allocating memory for the new node.
2. Storing the data in the node.
3. Updating the pointers to link the node correctly.

Similar to arrays, nodes in a linked list can be inserted at any position—at the start, at the end, or between existing nodes. Therefore, insertion can occur in one of the following three ways:

- ◆ Inserting at the beginning of the linked list.
- ◆ Inserting at the end of the linked list.
- ◆ Inserting at a particular position within the list.

3.2.1.1 Inserting a node at the Beginning

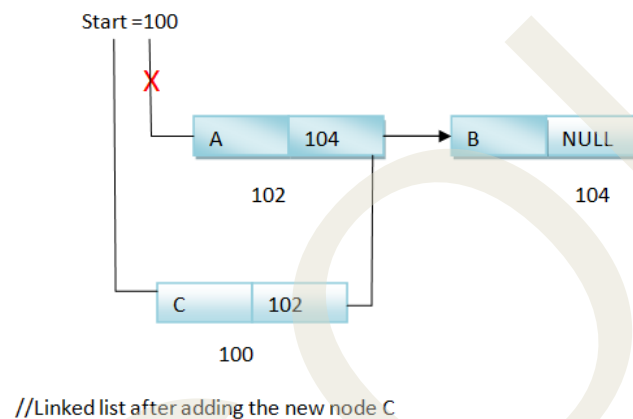
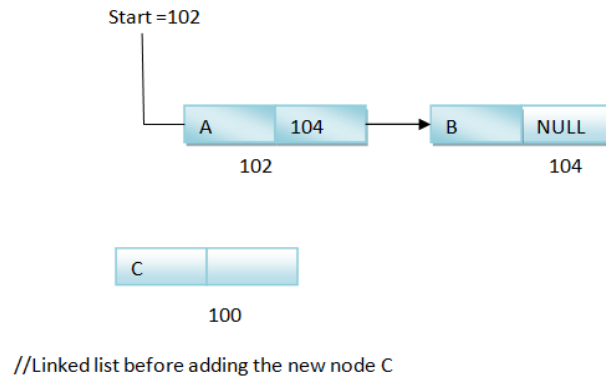
In order to insert a new node at the beginning of the linked list, the following steps are to be followed.

Step 1: If the linked list is empty or the new node needs to be inserted before the current first node, it will be placed at the beginning of the list.

Step 2: Create a new node and store the data in it.

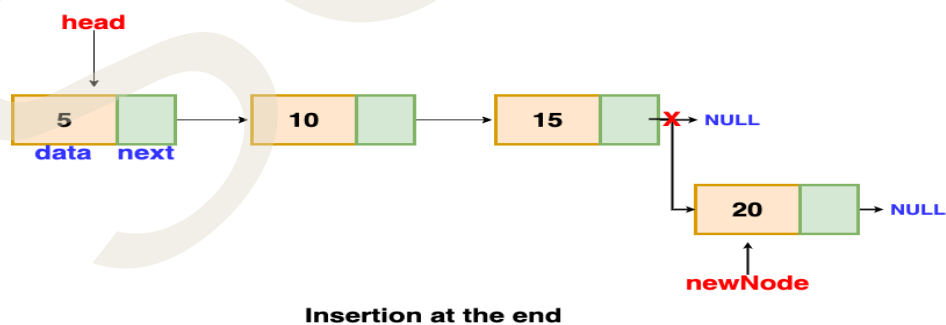
Step 3: Set the **link** (or pointer) part of the new node to point to the current first node (i.e., the node pointed to by Start or Head).

Step 4: Update the Start or Head pointer to point to the new node.



3.2.1.2 Inserting at the end of list

If the node to be inserted appears after the last node in the linked list then insert that node at the end of the linked list. For inserting a node at the end of the list, we have to copy the address of the new node to the link part of the last node. Assign NULL in the link part of the newly added node.



Step 1: Initialize the head pointer and set it to NULL.

Step 2: Create a new node with the given data and set its next pointer to NULL, since it will be the last node in the list.

Step 3: If the linked list is empty (i.e., head is NULL), assign the new node as the head of the list.

Step 4: If the list already contains nodes, traverse to the end of the list and link the last node's next pointer to the new node.

3.2.1.3 Inserting at specific location in the list (in between two nodes)

The following steps are used to insert a new node in a specific location in the linked list. Here we are trying to insert a node in the third position in the linked list. A series of operations to be performed to insert at the third position in a linked list. Let temp, pre-node and post-node are pointers of Node type structure. Assume POS is a variable which contains the value of the position where the node is to be inserted. The following steps can be developed for the insertion operation.

Step 1: Create a node and the address is stored in temp.

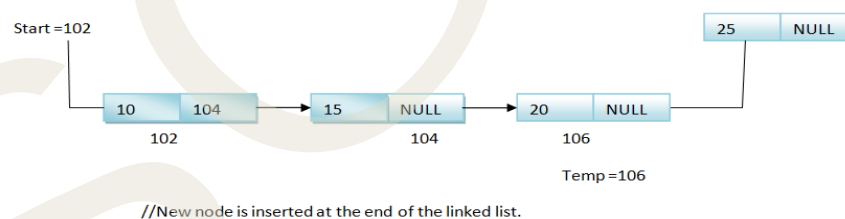
Step 2: Store the data and link part of this node using temp.

Step 3: By traversing the list, identify the location where the new node is to be inserted. Also obtain the address of the nodes at position POS-1 and POS in the pointers pre-node and post-node respectively (POS1 and POS are the positions of pre-node and post-node).

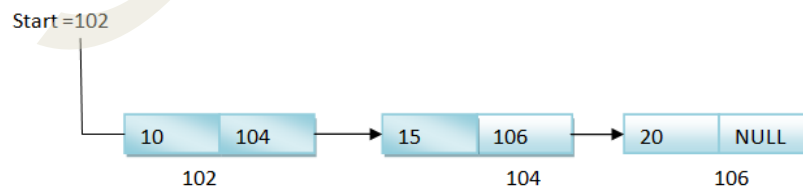
Step 4: Copy the content of the temp (address of the new node) into the link part of the node at position (POS-1), which can be accessed using prenode

Step 5: The content of postnode is copied (address of the node at position POS) to the link part of the new node which is pointed to by temp.

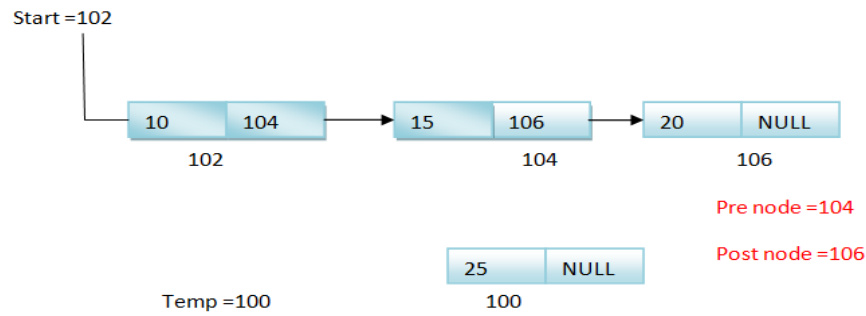
1) Linked list having three nodes. A new node is going to be inserted at the 3rd position



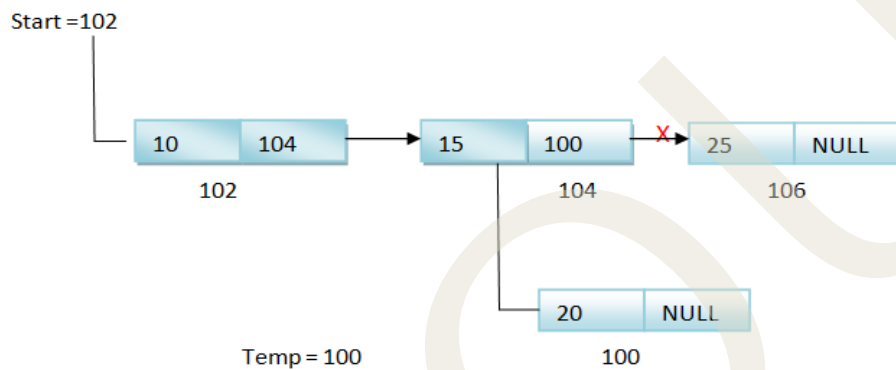
2) A new node is created and the address is stored in temp.



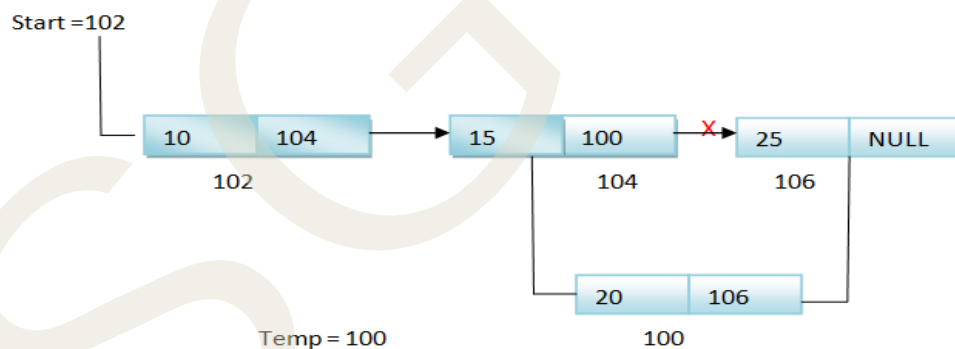
3) Address of the second node is copied into the prenode and the address of the third node is copied into the postnode.



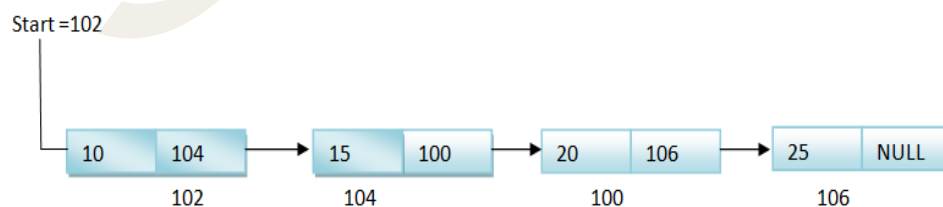
4) Address of the new node (from temp) is copied into the link part of the second node pointed to by the prenode and the new node is linked.



5) Address of the fourth node available in the postnode is copied into the link list part of the new node.



6) Linked list after the insertion of a new node at the third position.



3.2.2 Deletion from a linked list

Deleting a node from a linked list involves removing a specific node from the structure. The position of the node to be deleted is usually provided. If only the data value is given, the node containing that value must first be located, and its position determined before performing the deletion. Nodes can be removed from any position—beginning, end, or a specific location within the list. To delete the first node, the Start pointer is updated to point to the second node (i.e., the link part of the first node). To remove the last node, the next pointer of the second-last node is set to NULL.

3.2.2.1 Deleting from the beginning of the list

Step 1: Check whether list is empty (Start == NULL)

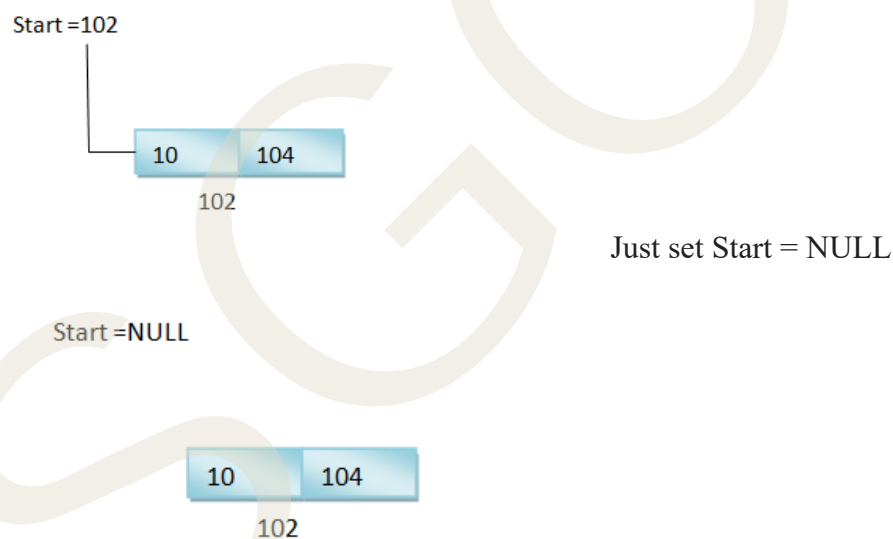
Step 2: If it is empty then, display 'List is Empty!!! Can't delete the node' and terminate the function.

Step 3: If it is not Empty then, check whether list is having only one node

Step 4: If it is true then set Start=NULL (Setting Empty list conditions) and stop.

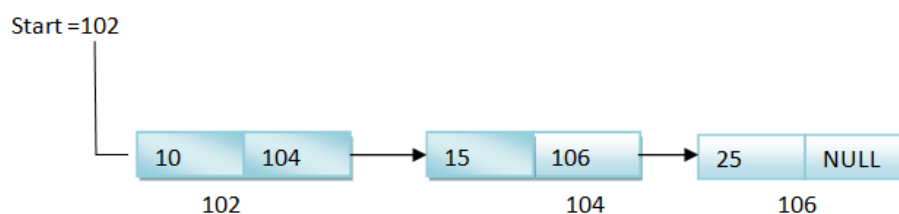
Step 5: If the list contains more than one node, then copy the content in the link part of the first node into Start.

1. List contains only one node.

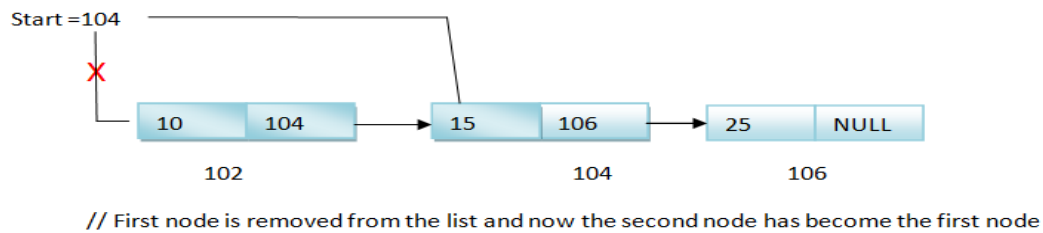


The node is removed from the list

2. List contains more than one node



Copy the content in the link part of the first node into Start.



First node is removed from the list and now second node has become the first node. The value of Start is now 104.

3.2.2.2 Deleting from End of the List

Step 1: Check whether the list is empty (Start==NULL).

Step 2: If it is empty then, display 'Empty list...!!!Cannot Delete' and terminate the function

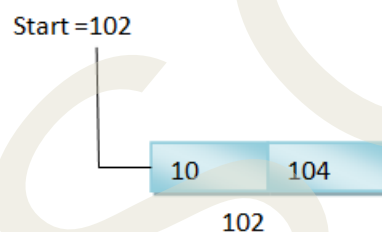
Step 3: If it is not Empty then, check whether list is having only one node

Step 4: If it is true then set Start=NULL (Setting Empty list conditions) and stop.

Step 5: If the list has more than one node, then traverse until the last node is reached. (Keep the address of the previous node in pre node while traversing).

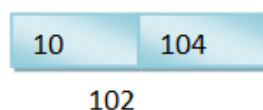
Step 5: Set NULL value in the link field of the pre node. Then the last node gets deleted from the list.

1. List contains only one node.



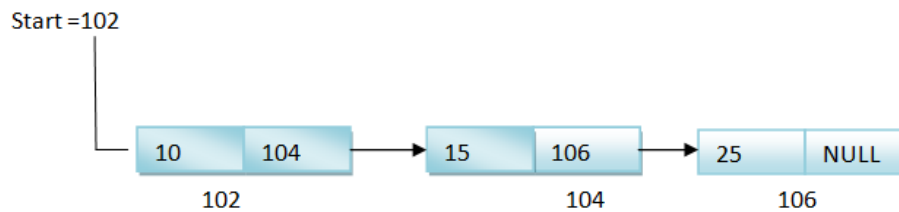
Just set Start = NULL

Start = NULL

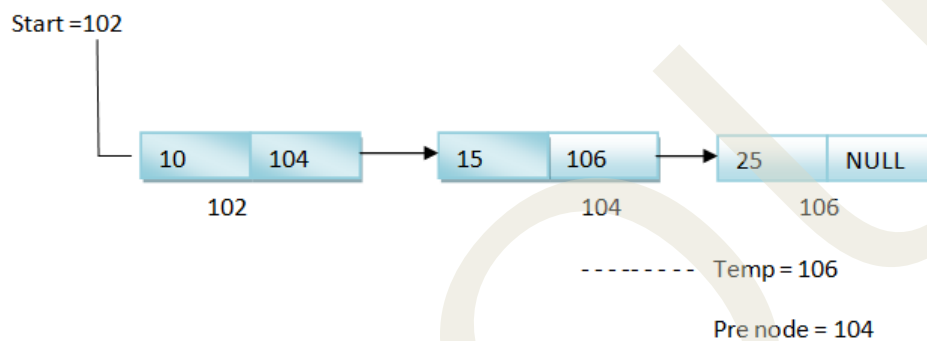


node is removed from the list

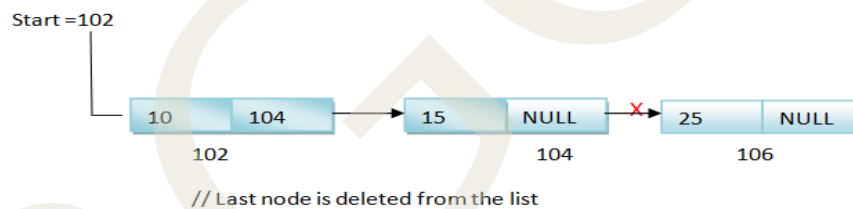
2. List contains more than one node



Traverse until the last node is reached. (Keep the address of the previous node in pre node while traversing).

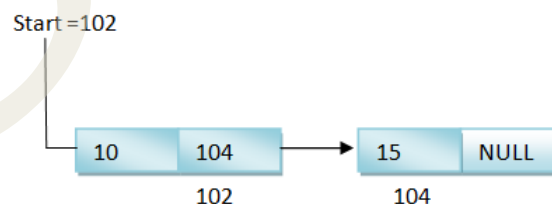


Set NULL value in the link field of the pre node



Last node is deleted from list

Linked list after the last node is deleted



3.2.2.3 Delete a node in specific location from the list

To remove a node from a specified position these all steps are to be performed. Below figure describes the procedure for the removal of the third node from the linked list having four nodes initially. it is assumed that prenode and postnode are pointers of Node type structure. Consider POS as a variable where the variable contains the position of

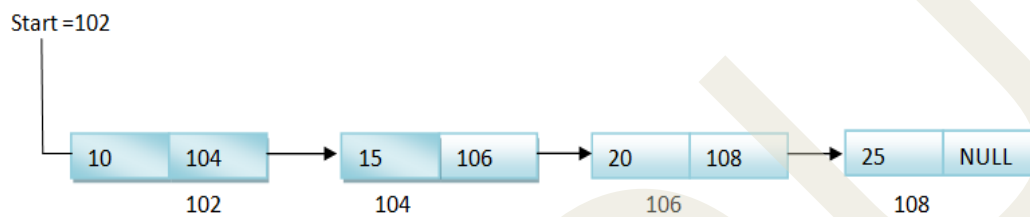
the node to be removed. Following steps are involved in the deletion operation.

Step 1: Obtain the address of the nodes at the position, POS1 and POS+1 in the pointers pre-node and post-node respectively, with the help of a traversal operation.

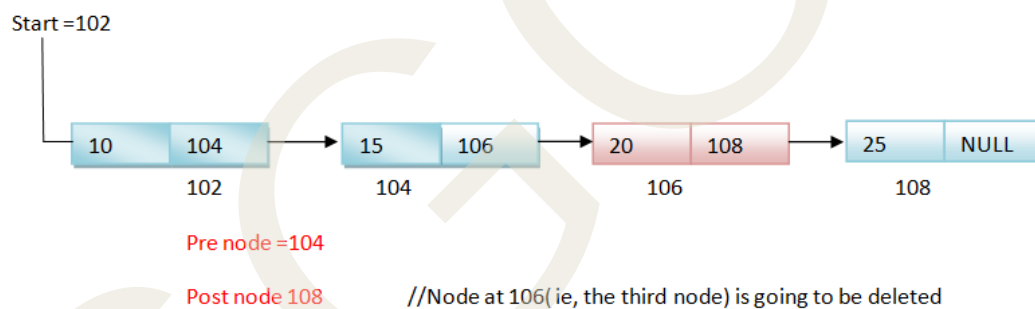
Step 2: Copy the content of postnode (address of the node at position POS+1) into the link part of the node at position (POS1), which can be accessed using prenode.

Step 3: Free the node at position POS.

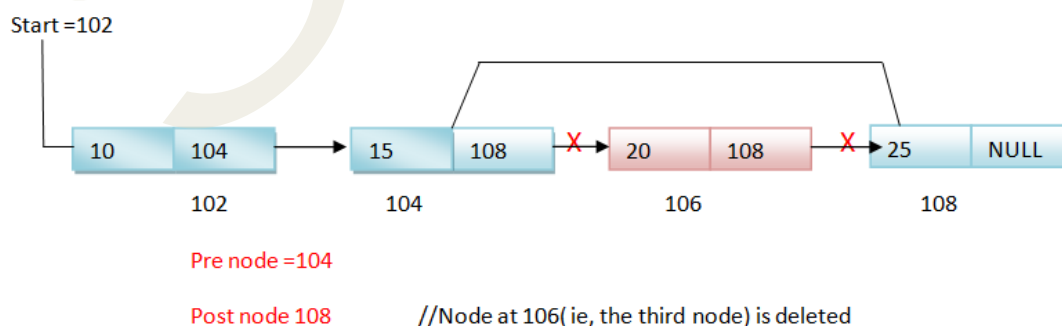
1) Linked list having four nodes



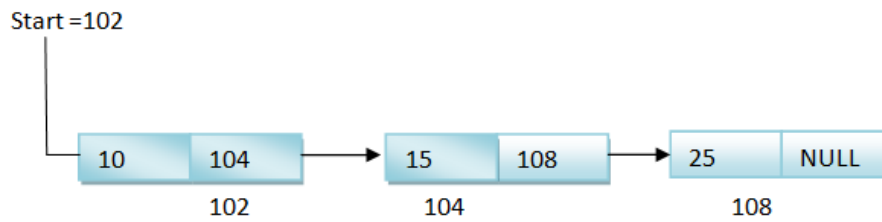
2) Address of the second node is stored in the prenode and the address of the fourth node is in postnode



3. Address of the fourth node available in post-node is copied into the link part of the second node pointed to by the prenode. Thus, the third node is removed from the list.



4. Linked list after the deletion of the third node.



If only the first two steps of deletion are performed in a linked list - such as updating pointers - the third node may still remain in memory and continue pointing to the fourth node. Therefore, it is important to explicitly release the memory occupied by the deleted node using the deallocation method provided by the programming language. Additionally, any temporary pointers used during linked list operations, such as temp, prenode, or postnode, should also be properly freed after completing the operation to avoid memory leaks.

3.2.3 Searching in a Linked list

Searching in a linked list involves traversing through the nodes of the list to find a specific value.

Searching in a Singly Linked List

Step 1: Start from the head (or Start) node.

Step 2: While the current node is not NULL:

- ◆ Access or process the data in the current node.
- ◆ Move to the next node using the link (next) pointer.

Step 3: Stop when the end of the list is reached (i.e., when the pointer is NULL).

3.2.4 Sorting in Linked list

Sorting a linked list involves organizing its elements in a specific order, such as ascending or descending (especially for numeric data). Various sorting algorithms can be applied to a linked list. In this case, we are focusing on **selection sort**. The core concept of selection sort is to repeatedly locate the smallest element in the unsorted portion of the list and move it to its correct position in the sorted portion.

Step 1: Initialize the sorted and unsorted parts of the list.

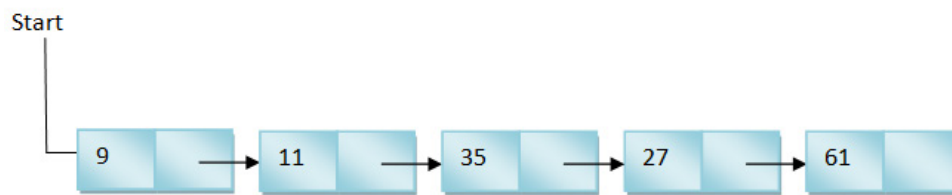
Step 2: Find the minimum element in the unsorted part.

Step 3: Move the minimum element to the end of the sorted part. That is, Swap the minimum element with the element at the end of the list for the first iteration, with the second last element in the second iteration and so on.

Step 4: Repeat until the entire list is sorted.

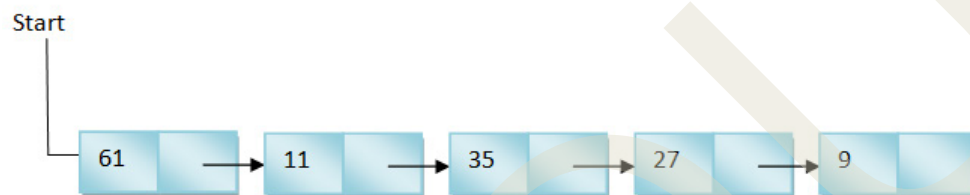
Example

Consider the list = [9 11 35 27 61]



We have to sort this list in descending order. For that, perform the following steps:

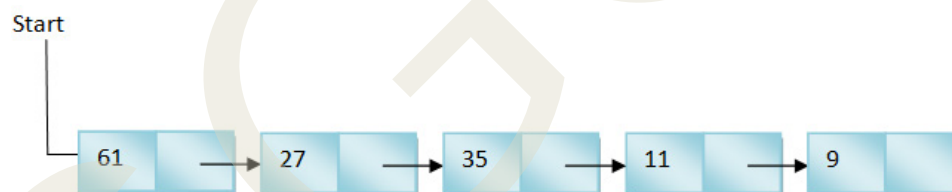
1. Find the minimum element in list (from 9 to 61) and swap it with the element at the last position.



// 9 and 61 are swapped

Now the element at the last position (9) is sorted and placed in its right position.

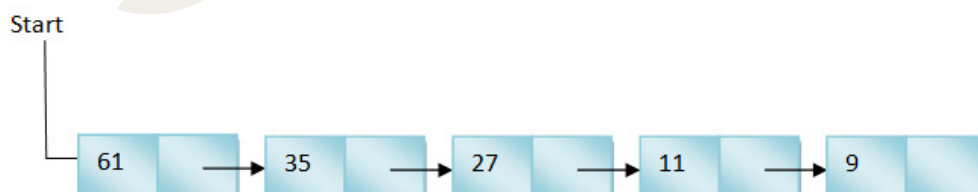
2. Find the minimum element in list (61 to 27) and swap it with the second last element (27).



// 11 and 27 are swapped

Now 9 and 11 are sorted and placed in their right position.

Find the minimum element from the remaining list (from 61 to 35). It is 27. So, swap the element at the third last element, i.e., 35.



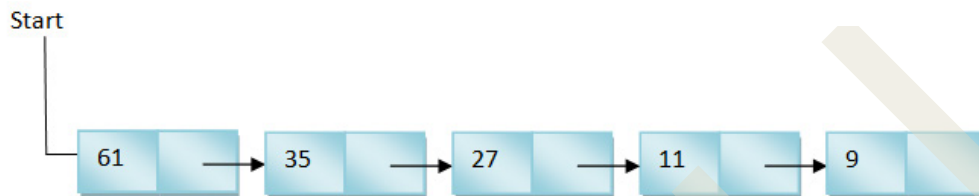
5. Find the minimum element from the remaining list (from 61 to 35). It is 35. Here 35 is to be swapped with 35 itself.

61 35 27 11 9

6. Now, only one element is remaining and it is in its right position.

61 35 27 11 9

The linked list after sorting is



3.2.5 Comparison between Array and Linked List

Table 3.2.1 Comparison Table

Array	Linked List
Arrays are stored in contiguous memory locations	Linked lists are stored in non-contiguous memory locations
Fixed in size; cannot grow or shrink after declaration	Dynamic in size; can grow or shrink during execution
Memory is allocated at compile time (static allocation)	Memory is allocated at run time (dynamic allocation)
Uses less memory, as it stores only data	Uses more memory because each node stores both data and a pointer to the next node
Elements can be accessed directly using an index.	Elements must be accessed sequentially by traversing the list.
Insertion and deletion are slower due to shifting of elements.	Insertion and deletion are faster (especially at the beginning or middle)

3.2.6 Applications of Linked list

1. Polynomial Representation

Linked lists can store each term of a polynomial in a node, making it easy to add, subtract, or multiply polynomials.

2. Arithmetic on Large Numbers

Linked lists can be used to perform operations like addition or subtraction on very large numbers by storing each digit in a separate node.

3. Implementation of Stacks and Queues

Linked lists can be used to build stacks and queues efficiently, allowing easy insertion and deletion without shifting elements.

4. Graph Representation (Adjacency List)

In graphs, linked lists are used to store the list of adjacent vertices for each node. This method is called an adjacency list and saves memory in sparse graphs.

5. Dynamic Memory Use

Linked lists grow and shrink as needed, so they are good for managing memory in applications where the number of elements changes frequently.

Recap

- ◆ A linked list is a dynamic data structure made of nodes, each containing data and a pointer to the next node.
- ◆ Unlike arrays, linked lists do not require contiguous memory and can grow or shrink during runtime.
- ◆ Insertion in a linked list involves allocating memory, storing data, and updating pointers; it can happen at the beginning, end, or any position.
- ◆ Deletion requires updating pointers and freeing memory to prevent leaks, whether removing the first, last, or a specific node.
- ◆ Searching a linked list means traversing nodes from the head until the desired value is found or the list ends.
- ◆ Sorting can be done using algorithms like selection sort by repeatedly finding the minimum element and repositioning it.
- ◆ Linked lists use more memory than arrays because of the pointers but allow faster insertions and deletions.
- ◆ Arrays have fixed size and support direct access, while linked lists offer dynamic size but need traversal for access.
- ◆ Linked lists are well-suited for applications where frequent insertions and deletions occur, like managing a music playlist dynamically.

Objective Type Questions

1. What type of data structure is a linked list?
2. In a linked list, what is stored along with data in each node?
3. What kind of memory allocation does a linked list use?
4. Which operation involves adding a node to a linked list?
5. Which operation involves removing a node from a linked list?
6. What is the pointer called that refers to the first node in a linked list?
7. Which operation involves finding a node with a specific value?
8. What does the last node's pointer in a singly linked list usually contain?
9. Compared to arrays, linked lists allow faster _____ and deletion.
10. Is the size of a linked list fixed or dynamic?

Answers to Objective Type Questions

1. Linear
2. Pointer
3. Dynamic
4. Insertion
5. Deletion
6. Head
7. Searching
8. NULL
9. Insertion
10. Dynamic

Assignments

1. Analyze why linked lists use more memory than arrays.
2. Discuss the impact of dynamic memory allocation on linked list performance
3. Describe the steps involved in inserting a node at the beginning of a linked list.
4. Explain how deletion of a node from the end of a linked list is performed.
5. Describe how searching for an element in a linked list is carried out

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
2. Lafore, R. (2002). *Data structures and algorithms in Java* (2nd ed.). Sams Publishing.
3. Weiss, M. A. (2013). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.
4. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.
5. Knuth, D. E. (1997). *The art of computer programming, Volume 1: Fundamental algorithms* (3rd ed.). Addison-Wesley.

Suggested Reading

1. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
2. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.
3. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (2009). *Data structures using C and C++* (2nd ed.). Pearson.
4. Lafore, R. (2001). *Object-oriented programming in C++* (4th ed.). Sams Publishing.
5. Kanetkar, Y. (2013). *Data structures through C* (2nd ed.). BPB Publications.

Unit 3

Linked List Representation of Stack and Queue

Learning Outcomes

After completing this section, learners will be able to:

- ◆ understand how stacks and queues work using linked lists.
- ◆ learn to perform push and pop operations in a linked stack.
- ◆ explore enqueue and dequeue operations in a linked queue.
- ◆ identify overflow and underflow conditions in linked implementations.
- ◆ familiarize with traversal techniques in stack and queue structures.

Prerequisites

You've already mastered the fundamentals of linked lists—how to insert, delete, and traverse nodes. Now it's time to take that understanding further and apply it in new, exciting ways. As you move ahead, you'll see how your prior knowledge becomes the foundation for building more dynamic and purposeful structures.

This next step will challenge your thinking and sharpen your skills in organizing data more effectively. You'll begin to appreciate how simple changes in how you link nodes can lead to powerful outcomes. Get ready to explore new patterns and strategies that will make your code smarter and more efficient.

Key words

Stack, queue, top, front, rear, enqueue, dequeue



Discussion

3.3.1 Stack Implementation Using Linked-List

We know that linked lists use dynamic memory allocation. When we implement a Stack using a linked list, the nodes are stored in non-contiguous memory locations. Each node has a pointer that links it to the next node in the Stack.

In this implementation, every new element is added at the top of the Stack. This means the top always points to the most recently added node. When we remove an element, we delete the node currently pointed to by top and then update top to point to the previous node in the list. The next field of the bottom-most node (the first element added) is always NULL.

Stack overflow happens in this case if there is not enough free memory in the heap to create a new node.

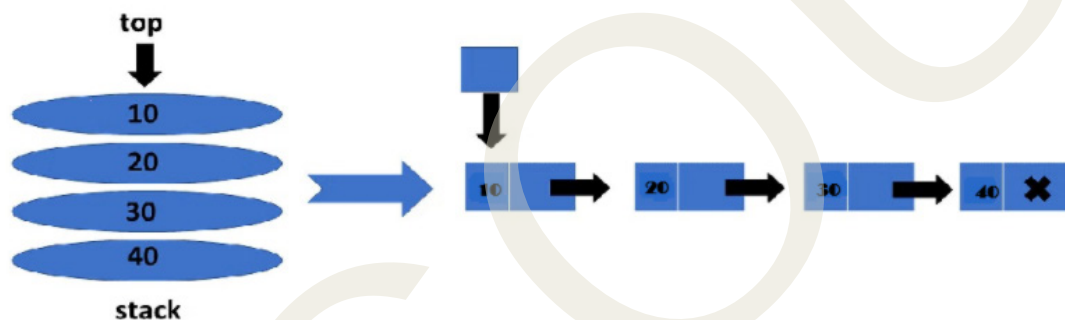


Fig.3.3.1 Linked list representation of stack

In the above fig.3.3.1, the top element is 10. The first element inserted is 40.

3.3.1.1 Push Operation

Adding a new node to the Stack is called a **push** operation.

Pushing a node in a linked list is different from inserting an element in an array. The push operation in a linked list-based stack involves the following steps:

1. First, create a new node and allocate memory for it.
2. If the list is empty, this new node becomes the first node. In this case, we assign the value to the data part and set the link (or next) part to NULL.
3. If the list already has nodes, the new node must be added at the beginning to maintain the Last In, First Out (LIFO) nature of the stack. To do this, assign the current top (or starting node) to the next field of the new node. Then update top to point to this new node.
4. An overflow condition happens when there's not enough memory available in the heap to create a new node.

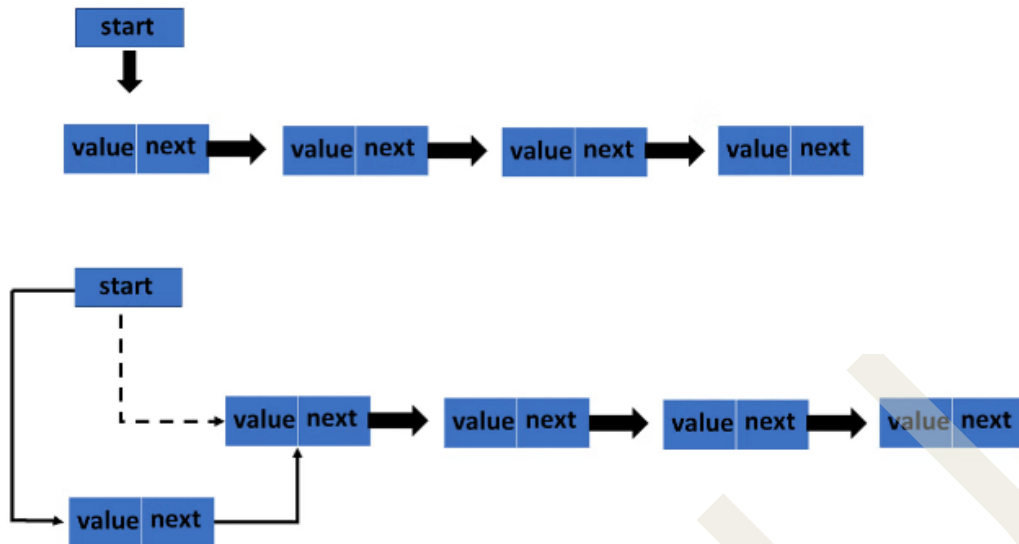


Fig 3.3.2 Push operation of stack using linked list

3.3.1.2 Pop Operation

Removing a node from the Stack is called a **pop** operation.

Popping a node from a linked list is different from popping an element from an array. The pop operation in a linked list-based stack involves the following steps:

5. In a Stack, the node is always removed from the top, which is the beginning of the linked list.
6. To remove the top node, we first store its data, then update the top pointer to point to the next node in the list. After that, the removed node is deleted from memory.
7. If the top pointer is NULL, it means the stack is empty. Trying to perform a pop operation in this case results in an underflow condition.

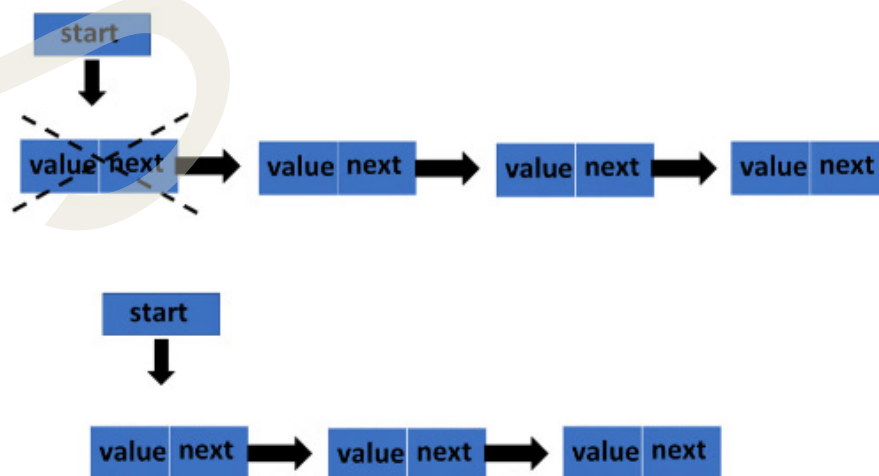


Fig 3.3.3 Pop operation using linked list

3.3.1.3 Traversing

Displaying all the nodes in a stack means going through each node of the linked list that represents the stack. This process is called traversal and involves the following steps:

1. First, copy the top pointer (or head pointer) to a temporary pointer.
2. Then, move the temporary pointer from one node to the next, and print the data stored in each node until you reach the end of the list

C program for implementing Linked Stack

```
#include <stdio.h>

#include <stdlib.h>

// Define a node

struct Node {

    int data;

    struct Node* next;

};

// Initialize top of the stack

struct Node* top = NULL;

// Push operation

void push(int value) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    if (!newNode) {

        printf("Heap Overflow\n");

        return;

    }

    newNode->data = value;

    newNode->next = top;

    top = newNode;
```

```

    printf("%d pushed to stack\n", value);
}

// Pop operation

void pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }

    struct Node* temp = top;

    top = top->next;

    printf("%d popped from stack\n", temp->data);

    free(temp);
}

// Peek operation

void peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
    } else {
        printf("Top element is %d\n", top->data);
    }
}

// Display stack

void display() {
    struct Node* temp = top;

    if (temp == NULL) {

```

```

    printf("Stack is empty\n");

    return;

}

printf("Stack elements: ");

while (temp != NULL) {

    printf("%d ", temp->data);

    temp = temp->next;

}

printf("\n");

}

// Main function to test stack operations

int main( ) {

    push(10);

    push(20);

    push(30);

    display( );

    peek( );

    pop( );

    display( );

    return 0;

}

```

3.3.2 Implementation of Queue using linked list

In a linked queue, each node of the queue consists of two fields, i.e., data field and reference field. Each entity of the linked queue points to its immediate next entity in the memory.

There are two key operations in a linked queue.

- ◆ Enqueue (Insertion) → Add an element at the rear. ie, Whenever a node is to be inserted, it is inserted only at the end of the linked list.
- ◆ Dequeue (Deletion) → Delete an element from front. ie, Whenever a node is to be deleted, only the first element is deleted from the linked list.

Let's us look at the steps in enqueue operation.

1. Create a new node.
2. Set the data and make its link part NULL.
3. If the queue is empty, set both front and rear to the new node.
4. If the queue is not empty, link the new node to the end and update rear to point to it.

Given below diagrams represent the enqueue operations of three elements 10,20 and 30.

Insert 10

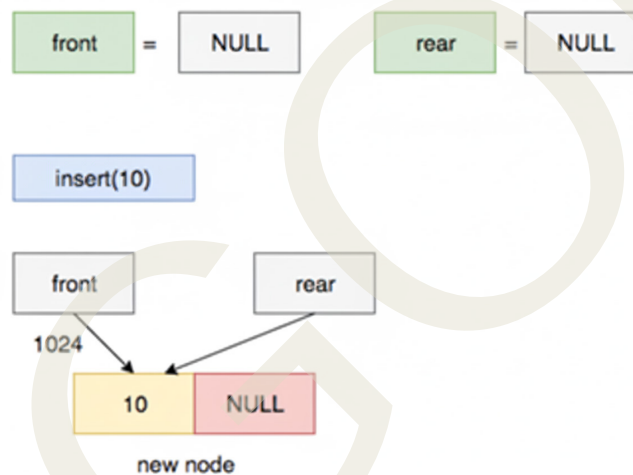


Fig 3.3.4 Enqueue value 10

Insert 20

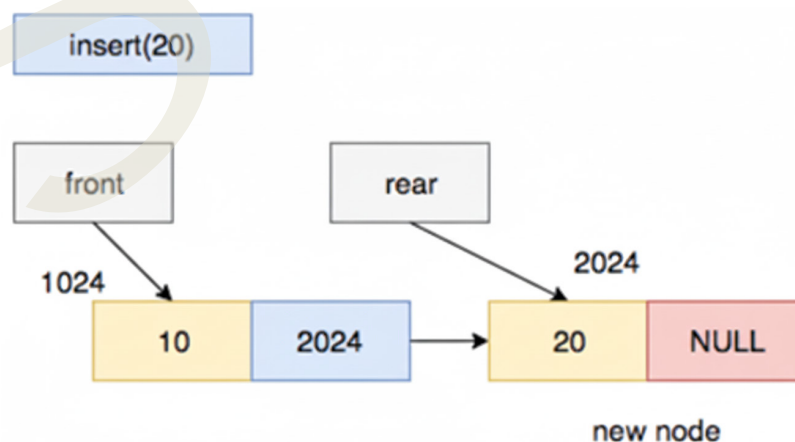


Fig. 3.3.5 Enqueue value 20

Insert 30

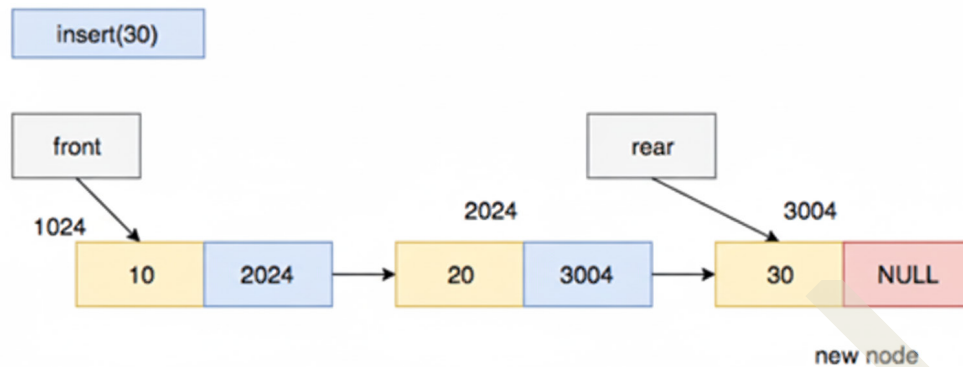


Fig 3.3.6 Enqueue value 30

Now we may look at how dequeue operation of queue using linked list

Here's the dequeue operation using a linked list explained step by step

- ◆ Check if the queue is empty; if it is, show underflow.
- ◆ Store the front node in a temporary pointer.
- ◆ Move the front to the next node.
- ◆ Free the memory of the removed node.
- ◆ If front becomes NULL, set rear to NULL.

Consider the linked list $10 \rightarrow 20 \rightarrow 30 \rightarrow 40$. Let us see which all elements are deleted in which order when dequeue operations are performed.

1. After first dequeue

$20 \rightarrow 30 \rightarrow 40$

2. After second dequeue

$30 \rightarrow 40$

3. After third dequeue

40

C program for implementing Linked Queue

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
// Node structure
```

```

struct Node {

    int data;

    struct Node* next;

};

// Queue front and rear

struct Node* front = NULL;

struct Node* rear = NULL;

// Enqueue operation

void enqueue(int value) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    if (!newNode) {

        printf("Memory allocation failed\n");

        return;

    }

    newNode->data = value;

    newNode->next = NULL;

    if (rear == NULL) {

        // Queue is empty

        front = rear = newNode;

    } else {

        rear->next = newNode;

        rear = newNode;

    }

    printf("%d enqueued to queue\n", value);

}

```

```

// Dequeue operation

void dequeue( ) {

    if (front == NULL) {

        printf("Queue Underflow\n");

        return;

    }

    struct Node* temp = front;

    front = front->next;

    // If queue becomes empty

    if (front == NULL) {

        rear = NULL;

    }

    printf("%d dequeued from queue\n", temp->data);

    free(temp);

}

// Display queue

void display() {

    if (front == NULL) {

        printf("Queue is empty\n");

        return;

    }

    struct Node* temp = front;

    printf("Queue elements: ");

    while (temp != NULL) {

        printf("%d ", temp->data);

```

```

        temp = temp->next;

    }

    printf("\n");

}

// Main function

int main() {

    enqueue(10);

    enqueue(20);

    enqueue(30);

    display();

    dequeue();

    display();

    enqueue(40);

    display();

    return 0;

}

```

3.3.3 Whether implementation of stack and queue using linked list or array is better?

Let's imagine you're running a ticket counter. Now, think of your array as a row of fixed chairs neatly arranged, all in a line. You know exactly how many people (or data) can sit there. It's fast and efficient, but if more people come in than you have chairs for, you're stuck. You'll have to bring in a whole new row, and sometimes even move people around to make room!

Now picture a linked list as a flexible queue that forms on the spot. People just hold hands and line up wherever there's space. No need to plan ahead for how many are coming, and if someone leaves or joins, it's easy to adjust without disturbing others.

When you're building stacks or queues, this difference really matters. If you want speed and simplicity, and you already know how many elements you're dealing with, arrays are great. But if your data structure needs to grow and shrink on the fly, or if you want to avoid the hassle of shifting things around, linked lists are like having a smart, adaptable helper.

So, it's not about which is better overall. It's about picking the right tool for the job, just like choosing between a fixed desk and a foldable table!

When Linked List is Better:

- ◆ **Dynamic Size:** Linked lists do not have a fixed size. You can grow or shrink the stack/queue as needed without worrying about overflow (unless memory is exhausted).
- ◆ **Efficient Insertions/Deletions:** In stacks and queues, elements are added and removed from one or both ends. Linked lists handle this efficiently without the need to shift elements, as arrays might require in queues.

When Array is Better:

- ◆ **Cache Friendly:** Arrays store data in contiguous memory locations, which makes access faster due to better cache performance.
- ◆ **Simplicity:** Array-based implementations are usually easier to code and understand for small, fixed-size problems.
- ◆ **Less Memory Overhead:** Unlike linked lists, arrays don't need extra space for pointers in each node.

Recap

Stack Implementation Using Linked List

- ◆ Linked lists use **dynamic memory allocation**.
- ◆ Nodes are stored in **non-contiguous memory**.
- ◆ Each node has a **data field** and a **pointer (next)** to the next node.
- ◆ New elements are always added at the top.
- ◆ **top** points to the most recently added node.
- ◆ Deletion is done from the top, and **top** is updated to point to the next node.
- ◆ Bottom-most node's **next** is always **NULL**.
- ◆ **Overflow** occurs when heap memory is full (no space for new node).

Push Operation (2.3.1.1)

- ◆ Create and allocate memory for a new node.
- ◆ If list is empty, node becomes first; **next** is **NULL**.
- ◆ If list is not empty, link new node to current **top**.
- ◆ Update **top** to the new node.

- ◆ Overflow occurs if memory allocation fails.

Pop Operation (2.3.1.2)

- ◆ Always remove node from the **top**.
- ◆ Store top node's data (if needed).
- ◆ Update **top** to point to the next node.
- ◆ Delete the previous top node.
- ◆ If **top** is **NULL**, stack is empty → underflow condition.

Traversing (2.3.1.3)

- ◆ Copy **top** to a temporary pointer.
- ◆ Move through nodes one by one using **next**.
- ◆ Print **data** of each node until **NULL** is reached

Queue Implementation Using Linked List

- ◆ Each node contains a data **field** and **reference (next)** field.
- ◆ Queue maintains two pointers: **front** and **rear**.
- ◆ Nodes point to the **next element** in memory.

Enqueue

- ◆ Insert node **only at the rear** (end of list).

Dequeue

- ◆ Delete node **only from the front** (beginning of list).
- ◆ Use **linked lists** if the size of the data structure is unpredictable or frequently changes.
- ◆ Use **arrays** if memory is limited, or if fast access and simplicity are priorities.

Objective Type Questions

1. What data structure is used to implement dynamic memory in stack and queue?
2. Which principle does a stack follow?
3. Which pointer in a stack points to the most recently added element?

4. In a linked stack, which operation adds an element?
5. In a linked stack, which operation removes an element?
6. What condition occurs when there is no space to create a new node?
7. What condition occurs when trying to pop from an empty stack?
8. Which pointer in a queue represents the start of the queue?
9. Which pointer in a queue represents the end of the queue?
10. Which operation inserts an element in a queue?
11. Which operation deletes an element from a queue?
12. What is the **next** field of the last node usually set to?
13. What type of memory allocation is used in linked list implementation?
14. Which traversal method is used to display all stack elements?
15. What is the term for nodes not being in contiguous memory?

Answers to Objective Type Questions

1. LinkedList
2. LIFO
3. Top
4. Push
5. Pop
6. Overflow
7. Underflow
8. Front
9. Rear
10. Enqueue
11. Dequeue
12. NULL

13. Dynamic
14. Linear
15. Noncontiguous

Assignments

1. Explain the structure and working of a stack using a linked list. Describe push and pop operations with appropriate diagrams.
2. Discuss how a queue can be implemented using a linked list. Illustrate enqueue and dequeue operations with diagrams.
3. Compare and contrast the implementation of stack and queue using linked list. Highlight their operations, memory usage, and advantages.
4. What are overflow and underflow conditions in linked stack and queue? Explain how these conditions are handled during operations.

Reference

1. Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
2. Horowitz, E., Sahni, S., & Mehta, D. (2008). *Fundamentals of Data Structures in C++* (2nd ed.). Universities Press.
3. Lafore, R. (2002). *Data Structures and Algorithms in C++* (2nd ed.). Sams Publishing.

Suggested Reading

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). *Data Structures and Algorithms in C++* (2nd ed.). Wiley.
3. Malik, D. S. (2010). *Data Structures Using C++* (2nd ed.). Cengage Learning.

Unit 4

Linked List Types

Learning Outcomes

After completing this section, learners will be able to:

- ◆ define a circular linked list and a doubly linked list.
- ◆ list the steps involved in creating a circular linked list.
- ◆ recall the algorithm for traversing a circular linked list.
- ◆ identify the structure of a node in a doubly linked list.
- ◆ state the basic steps for inserting and deleting a node in a doubly linked list.

Prerequisites

Before beginning this unit, it is helpful to recall what you already know about singly linked lists. In a singly linked list, each node contains some data and a pointer to the next node, and the last node points to NULL. This basic structure is useful for storing data in a linear and dynamic way. You may have encountered this while learning about how queues or stacks are implemented in memory.

In this unit, we will build on that knowledge and introduce two important types of linked lists: circular linked lists and doubly linked lists. These offer more flexibility than singly linked lists. For instance, in a circular linked list, the last node connects back to the first, forming a loop. This is useful in applications like a media player that plays songs in a continuous loop. In doubly linked lists, each node has two pointers, one to the next node and one to the previous node allowing movement in both directions. This is useful in applications like web browsers, where users can move back and forth between pages.

Understanding pointers, dynamic memory allocation, and basic linked list operations such as traversal, insertion, and deletion will help you grasp these advanced structures more easily.

Key words

Linked list, node, singly linked list, doubly linked list, circular linked list



Discussion

3.4.1 Circular Linked List

In the Singly Linked list, every node points to its next node in the sequence and the last node points to NULL. A circular Linked list is circular, which means it has no end. The last node points to the first node, creating a circle, hence the name. Media player that repeats endlessly where the last song points to the first song is an example of a Circular linked list. Also, in multiplayer games, all players are placed in a circular linked list. A circular linked list is similar to the singly linked list except that the last node points to the first node in the list.

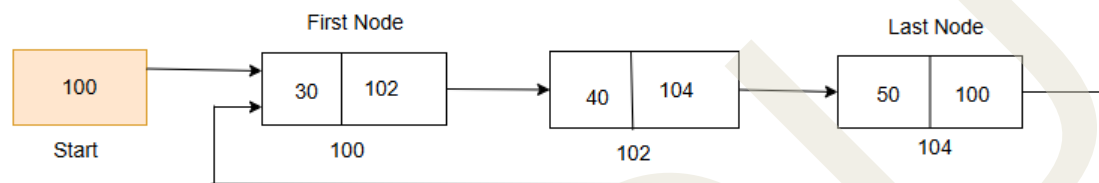


Fig 3.4.1 Circular Linked list

3.4.1.1 Creation of Circular linked list

The algorithm for creating a circular linked list is given below.

Step 1: Take a new node in the pointer called Start

Step 2: Read Data (Start)

Step 3: Take a pointer called End and point it to the same node being pointed by Start, i.e, Start= End.

Step 4: Bring a new node in the pointer called Temp

Step 5: Read Data (Temp)

Step 6: Connect the link part of End to Temp, i.e., End→link = Temp

Step 7: Set End = Temp

Step 8: Repeat steps 4 to 7 till the whole of the list is constructed

Step 9: Point Link of Temp to First

Step 10: Stop.

3.4.1.2 Traversing in Circular Linked List

The main advantage of a circular linked list is that from any node, one can reach any other node. Traversing in a circular linked list can be done by a loop. Initialise the temporary pointer variable Temp to Start pointer and run until the next pointer of Temp becomes Start. The algorithm is described as follows.

Algorithm

Step 1: The address of the first node is found from Start and stored in Temp.

Step 2: If the content of Temp = NULL, Stop

Step 3: else, using the address in Temp, get the data of the first or next node.

Step 4: The content of the link part of this node (i.e. the address of the next node) is stored in Temp

Temp = Temp → Link //value of temp is updated to link of temp.

Step 5: Repeat Step 4 and Step 5 until Temp → Link! = Start

//Till the last node, where the link part points to the start address

Step 6: Stop.

3.4.2 Structure of Doubly Linked List

A linked list is made up of a series of nodes connected through links. When we examine the linked list shown below figure 3.4.2, we can observe a key difference from a singly linked list: each node includes two links, a left link and a right link. This allows movement through the list in both directions, forward and backward. Such a linked list, where traversal is possible in both directions, is known as a doubly linked list.



Fig 3.4.2 Node of a doubly linked list

As shown in Fig. 2.4.2, a node in a doubly linked list is composed of three main parts:

1. **Data** – stores the required information.
2. **Left Link** – points to the previous node in the list.
3. **Right Link** – points to the next node in the list.

The left link of the first node is set to NULL, indicating there is no node before it. In the same way, the right link of the last node is NULL, since there is no node after it.

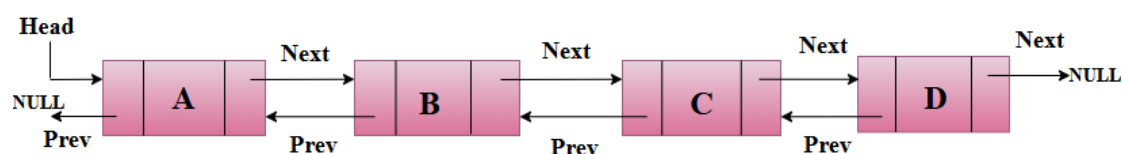


Fig 3.4.3 A Doubly linked list

The figure 3.4.3 illustrates a complete doubly linked list consisting of four nodes containing the data values A, B, C, and D. Each node is divided into three parts: the data field, a pointer to the previous node (Prev), and a pointer to the next node (Next). The list begins with a special pointer named Head, which points to the first node of the list. In this first node, the Prev pointer is set to NULL, indicating that there is no node before it. Each node is linked in such a way that the Next pointer of a node connects it to the following node ($A \rightarrow B \rightarrow C \rightarrow D$), while the Prev pointer connects it to the preceding node ($D \leftarrow C \leftarrow B \leftarrow A$), thus allowing traversal in both forward and backward directions. The last node in the list, which contains the data value D, has its Next pointer set to NULL, indicating the end of the list. This structure effectively demonstrates the bidirectional navigation feature of a doubly linked list.

3.4.3 The Algorithm for the Creation of a Doubly Linked List

The algorithm for the creation of a doubly linked list is given below:

Algorithm

- Step 1. Take a new node in the pointer called First.
- Step 2. Point leftLink of first to NULL, i.e., $\text{first} \rightarrow \text{leftLink} = \text{NULL}$.
- Step 3. Read Data(First).
- Step 4. Point back pointer to the node being pointed by First, i.e., $\text{back} = \text{First}$.
- Step 5. Bring a new node called Far to the pointer.
- Step 6. Read Data(Far).
- Step 7. Connect rightLink or back to Far, i.e., $\text{back} \rightarrow \text{rightLink} = \text{Far}$.
- Step 8. Connect leftLink of Far to Back, i.e., $\text{Far} \rightarrow \text{leftLink} = \text{Back}$.
- Step 9. Take back to Far, i.e., $\text{back} = \text{Far}$.
- Step 10. Repeat steps 5 to 9 till the whole of the list is constructed.
- Step 11. Point rightLink of far to NULL, i.e., $\text{Far} \rightarrow \text{rightLink} = \text{NULL}$.
- Step 12. Stop.

3.4.3.1 Insertion of a Node in Between Two Nodes in Doubly Linked List

Consider the following doubly linked list, which represents the insertion of a New Node in between two nodes, as shown in Figure 3.4.4. We are given a pointer to a node as prev node, and the new node is inserted after the given node. This can be done using the following steps:

- Step 1: First, create a new node (say, new node).
- Step 2: Now insert the data in the new node.
- Step 3: Point the next of the new node to the next of prev node.

Step 4: Point the next of prev_node to new_node.

Step 5: Point the previous of new node to prev_node.

Step 6: Point the previous of next of new node to new_node.

The following program segment can do insertion of New Node:

```
new_node->data = new_data
new_node->next = prev_node->next
prev_node->next = new_node
new_node->prev = prev_node
if (new_node->next != NULL)
    new_node->next->prev = new_node
```

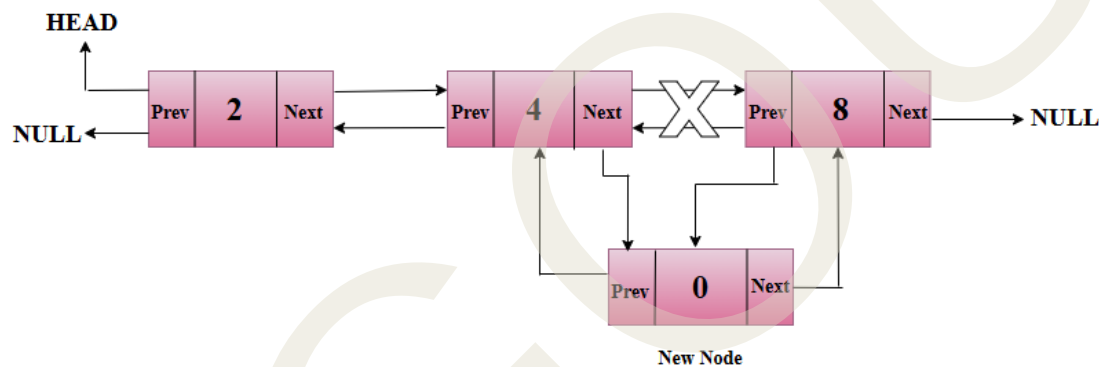


Fig 3.4.4 Insertion of new Node in between two nodes

3.4.4 Deletion of a node in doubly linked list

To delete a node from a doubly linked list at a specific position as illustrated in Figure 3.4.5, follow the steps below:

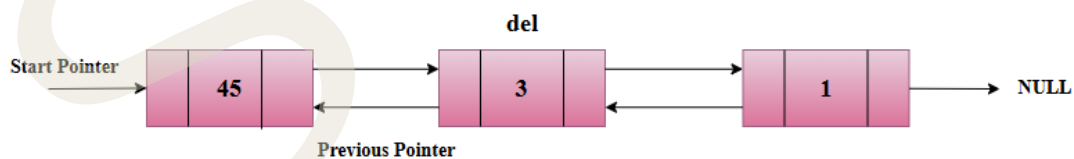


Fig 3.4.5 Deletion of a node in doubly linked list

Assume that the node to be removed is referred to as del.

- ◆ If del is the head node of the list, update the head pointer to refer to the next node.
- ◆ If del has a next node, update its prev pointer to refer to del's previous node.
- ◆ If del has a previous node, update its next pointer to refer to del's next node.

The following code segment performs the deletion operation in a doubly linked list:

```
if (del->next != NULL)
    del->next->prev = del->prev;
if (del->prev != NULL)
    del->prev->next = del->next;
free(del);
```

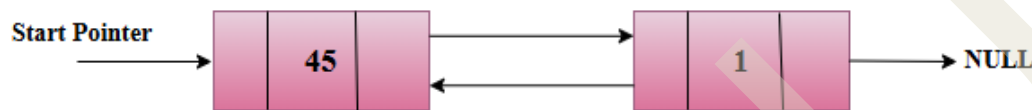


Fig 3.4.6 After the deletion of the middle node

Recap

Linked Lists:

- ◆ A linked list is a way to represent a sequence of elements.
- ◆ Each element is called a node.
- ◆ A node contains Data and a pointer called Next.
- ◆ The Next pointer links to the next node in the list, forming a chain.
- ◆ The list is pointed to by a pointer called ptr, and the last node's Next pointer points to NULL.

Singly Linked List:

- ◆ In a singly linked list, nodes are linked in one direction.
- ◆ Traversal is possible only in the forward direction.
- ◆ It's difficult to keep track of the previous node while traversing.

Doubly Linked List:

- ◆ Solves problems of singly linked lists by allowing traversal in both forward and backward directions.
- ◆ Each node contains two pointers: Next (points to the next node) and Prev (points to the previous node).

Circular Linked List:

- ◆ The last node's Next pointer links back to the first node, creating a loop.
- ◆ Unlike a singly or doubly linked list, the list doesn't have an endpoint.
- ◆ Circular linked lists are used in scenarios where continuous looping is required (e.g., media players, multiplayer games).

Objective Type Questions

1. Which pointer denotes the end of a singly linked list?
2. What is the pointer that connects the last node to the first node in a circular linked list?
3. In a doubly linked list, how many pointers does each node have?
4. Which type of linked list allows traversal in both directions?
5. In a circular linked list, the last node points to which node?
6. In a doubly linked list, which pointer points to the next node?
7. In a doubly linked list, which pointer points to the previous node?
8. What is the number of pointers affected by an insertion operation in a doubly linked list?

Answers to Objective Type Questions

1. NULL
2. Next
3. Two
4. Doubly
5. First
6. Right
7. Left
8. Two (Each node has only one pointer to traverse the list back and forth).

Assignments

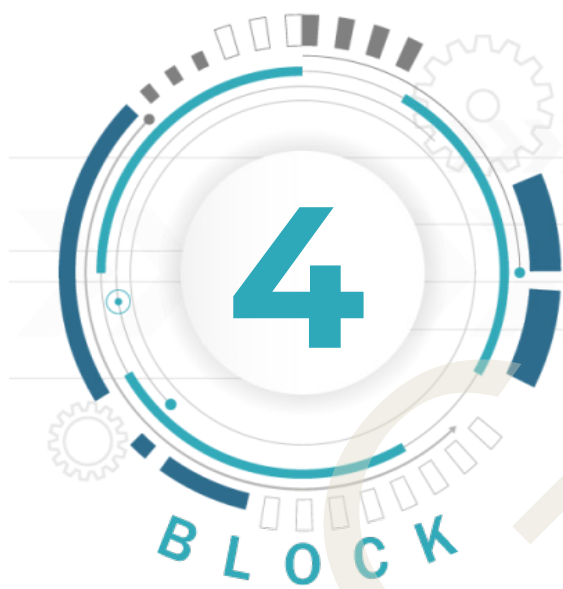
1. What is a Circular linked list? Explain the basic operations of a circular linked list.
2. Explain the algorithm for creating a doubly linked list.
3. Implement a doubly linked list with operations to insert and delete nodes at both ends.
4. Explain the difference between a singly linked list and a doubly linked list. Provide examples.
5. Write an algorithm to delete a node from a circular linked list and explain its steps.

Reference

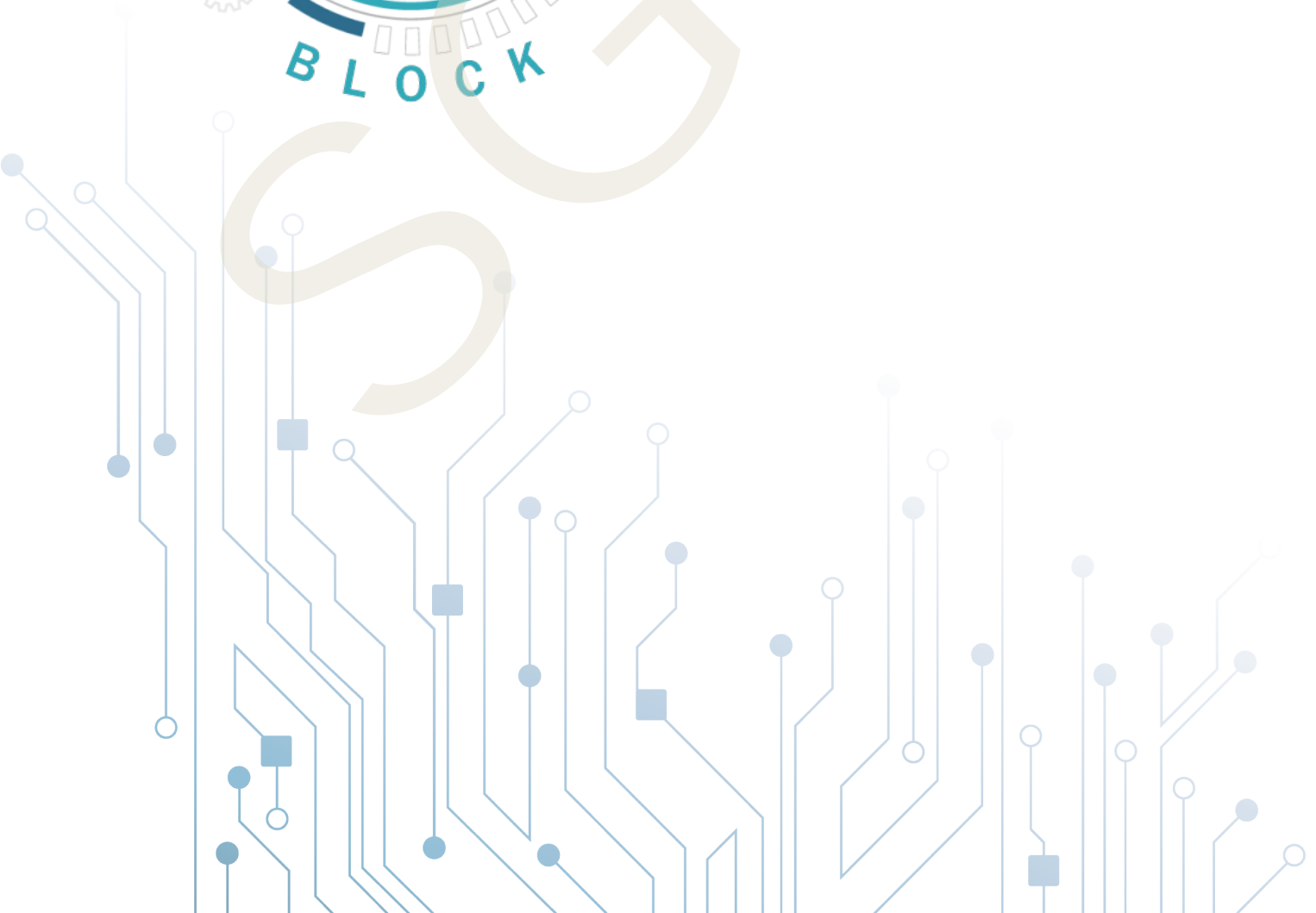
1. <https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-saman>

Suggested Reading

1. Sharma, A. K. "Data Structures using C", 2e. Pearson Education India, 2013.
2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and algorithms". Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
3. Weiss, Mark Allen. "Data structures and algorithm analysis". Benjamin-Cummings Publishing Co., Inc., 1995.



Non-Linear Data Structures



Unit 1

Trees

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ to learn about the concepts and terminologies of tree data structure.
- ◆ to study the concept of a binary tree and important types of binary tree
- ◆ to demonstrate the concepts of tree traversal algorithms

Prerequisites

In the previous units, we discussed arrays, stacks, queues, and linked lists, which are linear data structures. If we require ordered or sequential information, we use these linear data structures. In figure 4.1.1 is the classification of data structures shown below.

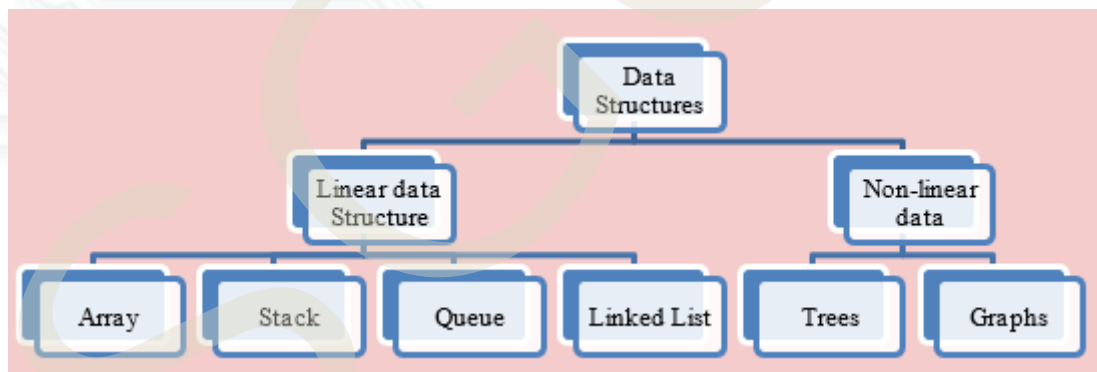


Fig 4.1.1 Classification of Data Structures

In linear data structure, data is stored in sequential manner. In this structure, every element has a unique predecessor and unique successor. In Figure 4.1.2 we can see a linear data structure. Here 1, 2, 3, 4 and 5 are stored in consecutive memory locations. The successor of 1 is 2 and the successor of 2 is 3 and so on. The predecessor of 2 is 1 and the predecessor of 3 is 2 and so on. Examples of these types of data structures are arrays, linked lists, stacks and queues.

In non-linear data structure the data is stored in a distributed manner. That is the data is stored in non-sequential form. So there is no previous or next element. Elements in

the non-linear data structure do not form a sequence. There is no unique predecessor or unique successor. In Figure 4.1.2 we can see a non-linear data structure. Here A, B, C, D and E are stored in non-consecutive memory locations and do not form a sequence. Examples are trees, graphs etc.

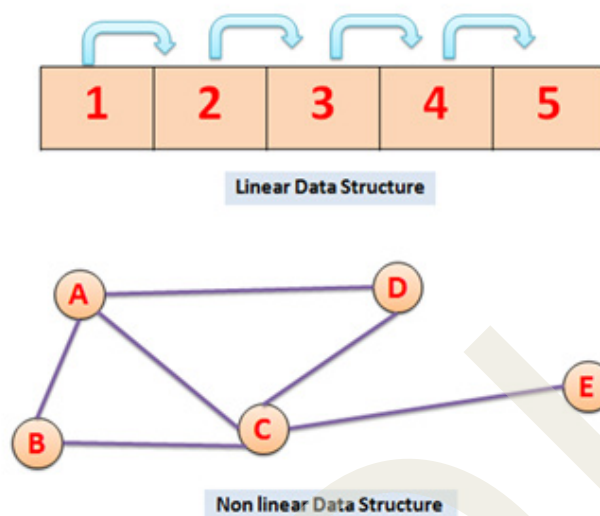


Fig. 4.1.2 Linear and Non-linear data structure

Trees and graphs come under non-linear data structures. Before studying tree data structures, binary trees, and tree traversal, students should understand basic data structures such as arrays, linked lists, stacks, and queues. They should be familiar with concepts of nodes, pointers, and dynamic memory allocation. A good grasp of recursion and basic algorithmic problem-solving techniques is also essential. Additionally, students should have basic programming skills to implement and manipulate these data structures effectively.

Key words

Binary Trees, Complete Binary Tree, 2-Tree or Extended Binary trees, Inorder traversal, Preorder Traversal and Postorder Traversal

Discussion

4.1.1 Introduction to Tree Concepts

Trees are hierarchical data structures consisting of nodes connected by edges. Each tree has a root node, which serves as the starting point, and every other node is connected by edges forming a parent-child relationship. Key terminologies include root, leaf, sibling, and subtree, which help in understanding the structure and relationships within a tree. The following figure 4.1.3 shows a tree structure. Each line connecting two nodes denotes a relation, namely parent-child relation between two nodes.

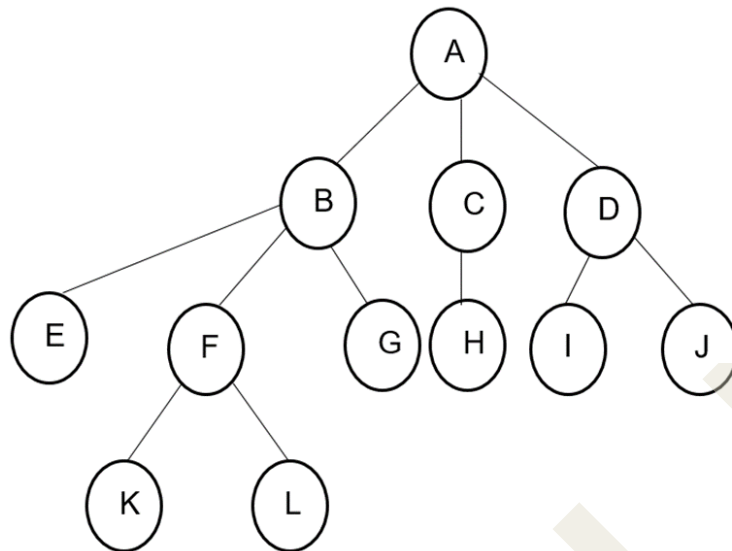


Fig 4.1.3 Structure of Tree

When a person is traveling from Cochin to Mumbai, there are many modes of transportation that he can take. This includes by sea, by road, by air, by rail. Following figure 4.1.4 shows the options available for travel from Cochin to Mumbai.

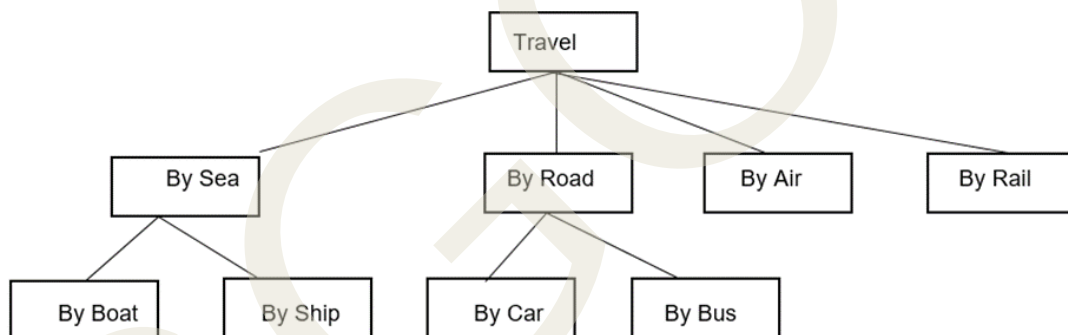


Fig 4.1.4 Options available for travel from Cochin to Mumbai

When a person is traveling by sea, he can use either a boat or a ship. Similarly, while traveling on the road, he can use either a car or bus. The node below a given node connected by its edge downward is called its child node. 'By Sea' and 'By Road', 'By Air' and 'By Rail' in figure 4.1.4 are called child nodes of 'Travel', the root node. A leaf node is a node without any children (Example: By ship, By Boat, By car, By Bus, By Air, By Rail). The root of the tree is the node with no parents. In figure 4.1.3, travel is the root node.

We can use another example for the tree data structure, i.e., a computer stores information about files and folders in a hierarchical manner in the memory. Figure 4.1.5 shows the root node "course" with subfolders named Mechanical, Computer, Electrical, and Electronics. The subfolders of Computer are B.Tech and BCA.

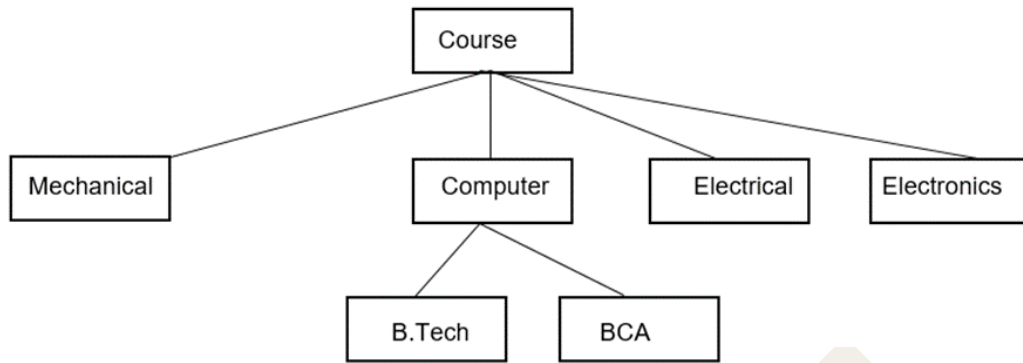


Fig 4.1.5 Example of file and folder hierarchy inside computer

4.1.1.1 Tree Key Terminologies

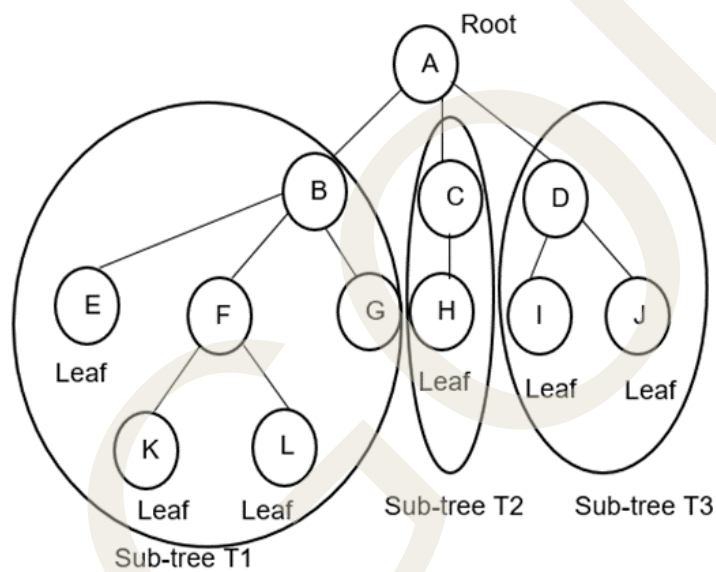


Fig 4.1.6 The Tree and its related terms

From figure 4.1.6, the following components of a tree can be identified.

- ◆ Root : A
- ◆ Child nodes of A : B, C, D
- ◆ Leaf nodes : E, G, H, I, J, K, L

Now let us discuss few terminologies associated with a tree:

- ◆ Node - A tree is a collection of entities called nodes. Nodes are connected by edges. Each node contains a value or data, and it may or may not have a child node.

Example: A, B, C...L in the figure 4.1.6.

- ◆ Root - The node which has no parent.

Example: A is the root of the tree given in the figure 4.1.6.

- ◆ Leaf - The node which has no children is called a leaf or terminal node.

Example: I, J, K, L, etc. are leaf nodes in the figure 4.1.6.

- ◆ Height - The number of nodes present in the longest path of the tree from root to a leaf node is called the height of the tree. This will contain a maximum number of nodes.

Example: One of the longest paths in the tree shown in figure 4.1.6 is A-B-F-K. Therefore, the height of the tree is 4.

- ◆ Depth - The depth of a node is the length of its path from the root node. The depth of the root node is taken as zero.

Example: The depths of nodes G and L are 2 and 3, respectively.

- ◆ Degree - Degree of a node is defined as the number of children present in a node. Degree of a tree is defined as equal to the degree of a node with maximum children.

Example: Degrees of nodes C and D are 1 and 2, respectively. Degree of the tree is 3 as there are two nodes A and B having maximum degree equal to 3.

4.1.2 Binary Trees

In a normal tree, each node can have any number of children. However, in a binary tree, each node can have at most two children, or it can be an empty tree. A binary tree is a finite set of nodes that either has no nodes or consists of a root node and two separate binary trees known as the left subtree and the right subtree. Figure 4.1.7 illustrates the generic structure of a binary tree.

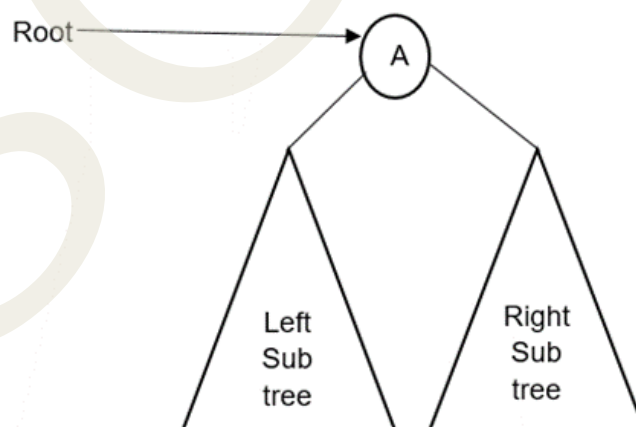


Fig 4.1.7 Generic Binary Tree

A non-empty binary tree consists of the following:

- ◆ A node called the root node

- ◆ A left sub-tree
- ◆ A right subtree

Binary tree are classified into two types:

1. Complete binary tree
2. Full binary tree

4.1.2.1 Complete Binary Trees

A **Complete Binary Tree** is a special type of binary tree in which all levels are completely filled except possibly the **last level**, and in the last level, all nodes are placed as left as possible. This structure ensures that the tree is as compact as it can be, with no gaps between nodes from left to right. It is especially useful in array-based representations of trees, such as **heaps**.

Key Properties of a Complete Binary Tree:

- ◆ All levels (except possibly the last) are **fully filled**.
- ◆ Nodes in the last level appear as far left as possible.
- ◆ It allows efficient storage in arrays (index-based representation).
- ◆ The height of a complete binary tree with n nodes is $\lfloor \log_2(n) \rfloor$.

The following figure 4.1.8 is a **complete binary tree** because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

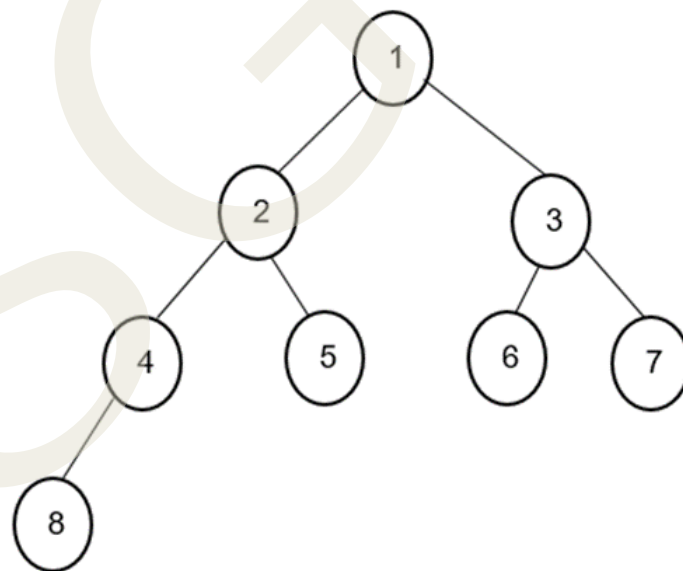
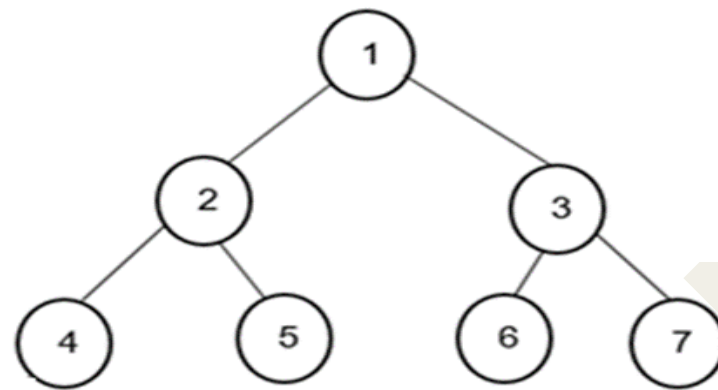


Fig 4.1.8 Complete Binary Tree

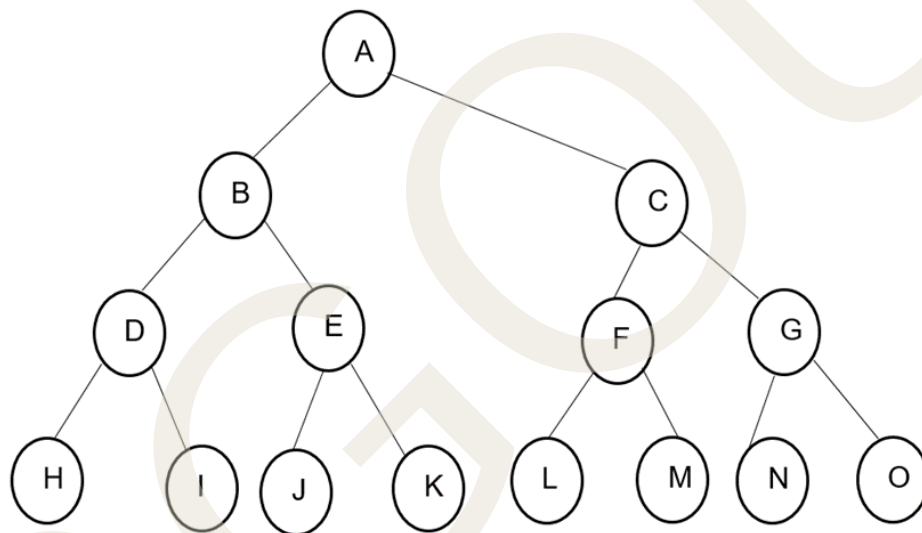
4.1.2.1 Full Binary Trees

A full binary tree contains the maximum allowed number of nodes at all levels. This means that each node has exactly zero or two children. The leaf nodes at the last level

will have zero children and the other non-leaf nodes will have exactly two children as shown in the following figure 4.1.9 (a) and (b).



(a)



(b)

Fig 4.1.9 Full Binary Trees

A complete binary tree is a full tree except at the last level where all nodes must appear as far left as possible.

Properties of a Full Binary Tree:

- ◆ Each internal node has exactly two children.
- ◆ All leaf nodes are at various levels, but each parent has two children.
- ◆ If there are n internal nodes, then the total number of nodes is $2n + 1$.
- ◆ The number of leaf nodes is $n + 1$.

4.1.2.3 2-Tree or Extended Binary Trees

An **Extended Binary Tree** (also called a 2-tree) is a binary tree where all the original

nodes with one or zero children are converted into nodes with exactly two children by adding special external or dummy nodes (also called null or sentinel nodes). It is mainly used in theoretical analysis of binary trees and expression trees. The below figure 4.1.10 is an example for extended binary trees. Here, all the nodes except G have two children. G has no children.

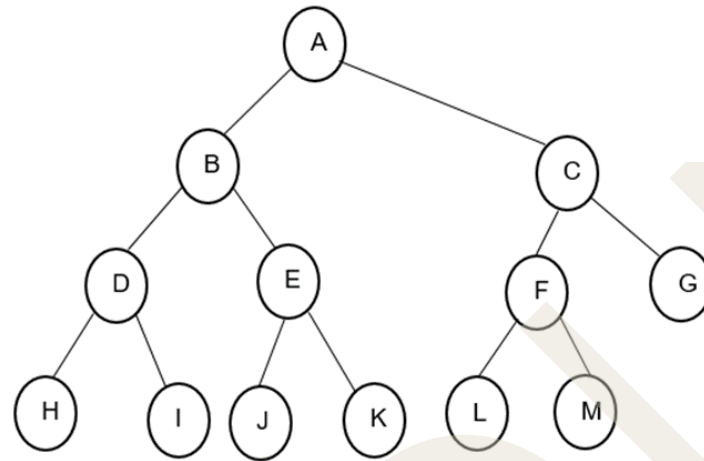


Fig 4.1.10 Extended Binary Tree

Characteristics of an Extended Binary Tree:

- ◆ All nodes have either 0 or 2 children, similar to a full binary tree.
- ◆ Leaf nodes (external nodes) are dummy or null nodes, not regular data nodes.
- ◆ Used for analyzing expression trees, threaded trees, or parsing trees.

4.1.3 Structure of a Node

In a tree, each element is called a node. A node is a fundamental building block of any tree-based data structure. Each node comprises the following:

1. It contains some information.
2. It has an edge to a left child node.
3. It has an edge to a right child node.

The above components of a node can be comfortably represented by a linked list as shown in Figure 4.1.11. Thus, a node consists of:

- a. pointer that points towards the right node(Right Child Address)
- b. pointer that points towards the left node(Left Child Address)
- c. data element.

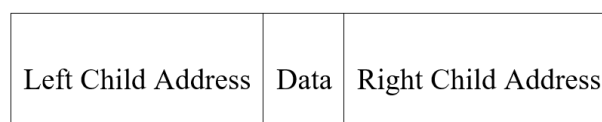


Fig 4.1.11 Node of a binary Tree

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes. In a tree, all nodes share common construct.

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node *rightChild;  
};
```

4.1.4 Tree Traversal

Suppose you have been given a task to do system maintenance at three offices in different destinations namely Cochin, Kozhikode and Thiruvananthapuram. You can do the maintenance in these cities in the following order.

1. Kozhikode, Cochin, Thiruvananthapuram
2. Kozhikode, Thiruvananthapuram, Cochin
3. Cochin, Thiruvananthapuram, Kozhikode
4. Cochin, Kozhikode, Thiruvananthapuram
5. Thiruvananthapuram, Cochin, Kozhikode
6. Thiruvananthapuram, Kozhikode, Cochin



Fig 4.1.12 Kerala Map

If you have a close look at the Kerala Map in figure 4.1.12, you will realize that Kozhikode is in the northern region of Kerala while Cochin and Thiruvananthapuram are in the southern region. Moreover, Cochin and Thiruvananthapuram are nearer to each other when compared to Kozhikode. Let us take Thiruvananthapuram as the nodal position, Cochin as the left and Kozhikode as right travel destinations. When we set a condition that the left would be visited before right, then the following three combinations are possible.

- A. Cochin, Thiruvananthapuram, Kozhikode
- B. Thiruvananthapuram, Cochin, Kozhikode
- C. Cochin, Kozhikode, Thiruvananthapuram

In the same manner, binary tree is also traversed for many purposes such as for searching a particular node, for processing some or all nodes of the tree and the like. (In the case of a binary tree, every node can have a maximum of 2 children.) There are three types of tree traversals. Option A in the above example is similar to inorder traversal where, left subtree traversal is followed by root followed by right subtree traversal. Option B in the above example is similar to a preorder traversal where, root followed by the left and right subtree is traversed. Option C in the above example is similar to postorder traversal where, left subtree traversal is followed by right subtree traversal that is followed by root visit.

During tree traversal, all the nodes of a tree are visited and their values may be printed. The reasons for binary tree traversal includes searching a particular node, for processing some or all nodes, etc. The following operations are possible on a node of a binary tree:

- (1) Process the visited node – V.
- (2) Visit its left child – L.
- (3) Visit its right child – R.

Any combinations of the above three operations can be done for tree traversal. The tree traversals have been named as preorder, inorder, and postorder according to the operation visit node (V).

4.1.4.1 Inorder Traversal

In the example given in the prerequisite, Cochin, Thiruvananthapuram, Kozhikode travel path is similar to inorder traversal where, left subtree traversal is followed by root followed by right subtree traversal. Let us go into the details of inorder traversal.

L-V-R : Inorder traversal, that is, travel the left sub-tree (L), process the visited node (V) and travel the right subtree (R).

Algorithm : An algorithm for inorder travel (L-V-R) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
inorderTravel(Tree){  
    if (Tree == NULL) return
```

```

else
{
    inorderTravel (leftChild (Tree));

    process DATA (Tree);

    inorderTravel (rightChild (Tree));
}
}

```

In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree i.e, call `inorderTravel (leftChild (Tree));`
2. Visit the root i.e; process `DATA (Tree);`
3. Traverse the right subtree, i.e., call `inorderTravel (rightChild (Tree));`

Illustration: Figure 4.1.14 shows the illustration of the Inorder traversal of the binary tree in figure 4.1.13.

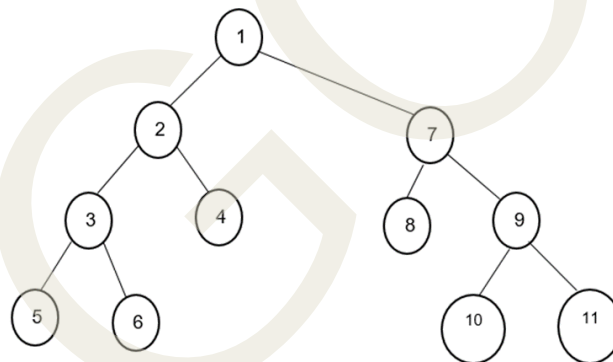


Fig 4.1.13 Binary Tree

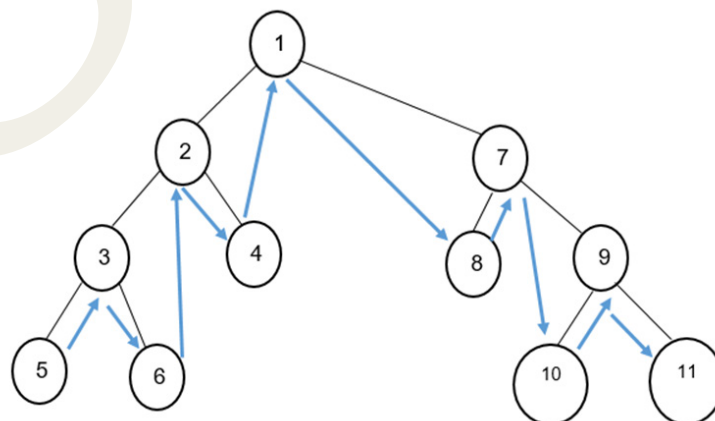


Fig 4.1.14 Inorder traversal

In case of figure 4.1.14, the Inorder traversal will give the following result.

5	3	6	2	4	1	8	7	10	9	11
---	---	---	---	---	---	---	---	----	---	----

Explanation: In the case of Inorder traversal, L is followed by V that is followed by R. So, in figure 4.1.13, first 5(L) is visited followed by 3(V) and then 6(R). This is followed by 2(V) which is followed by 4(R). Then 1(V) is visited, followed by 8(L) and then 7(V). This is followed by 10(L), 9(V) and 11(R).

4.1.4.2 Preorder Traversal

In the travel path example discussed above, the path B (Thiruvananthapuram, Cochin, Kozhikode) is similar to a preorder traversal where the root followed by the left and right subtree are traversed. Let us now go into the details of preorder traversal.

V-L-R : Preorder traversal, that is, process the visited node (V), travel the left sub-tree (L) and travel the right subtree (R)

Algorithm: An algorithm for preorder travel (V-L-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

```
preorderTravel(Tree){  
    if (Tree == NULL) return  
    else  
    {  
        process DATA (Tree);  
        preorderTravel (leftChild (Tree));  
        preorderTravel (rightChild (Tree));  
    }  
}
```

In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Visit the root i.e; process DATA (Tree);
2. Traverse the left subtree i.e, call preorderTravel (leftChild (Tree));
3. Traverse the right subtree, i.e., call preorderTravel (rightChild (Tree));

Illustration: Figure 4.1.15 shows the illustration of the Inorder traversal of the binary tree in figure 4.1.13.

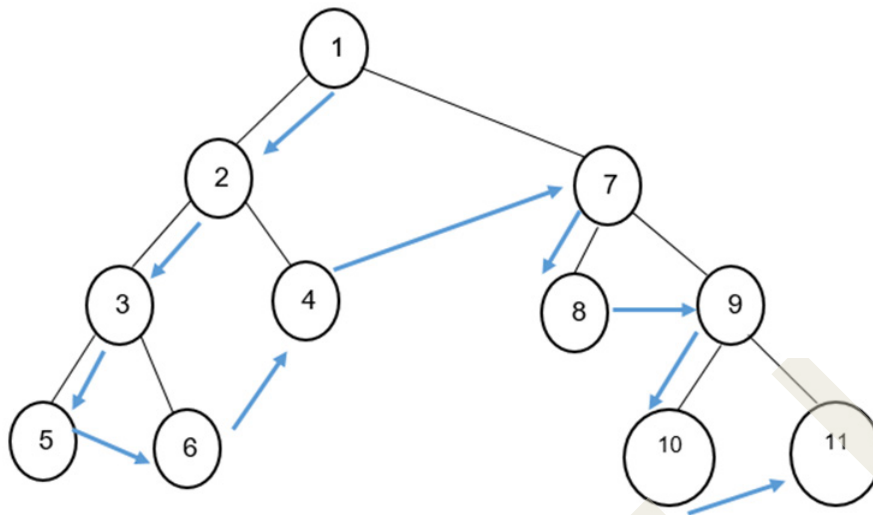


Fig 4.1.15 Preorder traversal

In case of figure 4.1.15, the Preorder traversal will give the following result.

1	2	3	5	6	4	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

Explanation: In the case of preorder traversal, node(V) is visited followed by L and R. So, in figure 4.1.14, 1(V) is visited followed by 2(L), 3(L), 5(L). then this is followed by 6 (R), 4(R). This is followed by 7(V) that is followed by 8(L). This is followed by 9 (V), 10(L), and 11(R).

4.1.4.3 Postorder Traversal

In the travel path example the path C - Cochin, Kozhikode, Thiruvananthapuram travel path is similar to postorder traversal where, left subtree traversal is followed by right subtree traversal that is followed by root visit.

Let us now go into the details of postorder traversal.

L-R-V : Postorder traversal, that is, travel the left sub-tree (L), travel the right sub-tree (R) and process the visited node (V)

Algorithm : An algorithm for postorder travel (L-R-V) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```

postorderTravel(Tree){
    if (Tree == NULL) return
    else
    {
        postOrderTravel (leftChild (Tree));
    }
}

```

```

    postOrderTravel (rightChild (Tree));

    process DATA (Tree);

}

}

```

In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree i.e, call postorderTravel (leftChild (Tree));
2. Traverse the right subtree, i.e., call postorderTravel (rightChild (Tree));
3. Visit the root i.e; process DATA (Tree);

Illustration : Figure 4.1.16 shows the illustration of the Inorder traversal of the binary tree in figure 4.1.13.

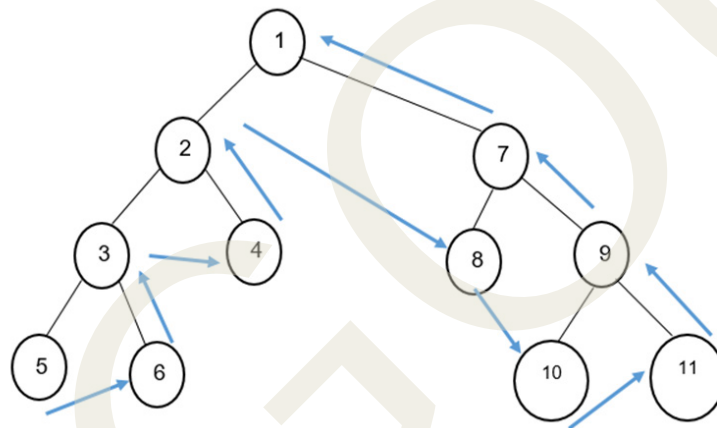


Fig 4.1.16 Postorder traversal

In case of figure 4.1.16, the Postorder traversal will give the following result.

5	6	3	4	2	8	10	11	9	7	1
---	---	---	---	---	---	----	----	---	---	---

Explanation: In the case of Postorder traversal, 5(L) is followed by 6(R) which is followed by 3(V). This is followed by 4(R) which is followed by 2(V). This is followed by 8(L), 10(L) and 11(R). This is followed by 9(V), 7(V), and 1(V).

As a summary, trees are a fundamental non-linear data structure that effectively represent hierarchical relationships in computing. Starting from the basic structure, each tree is composed of interconnected nodes, with special types like binary trees where each node has at most two children providing a structured way to store and access data. Variants such as **complete binary trees**, **full binary trees**, and **2-trees (extended binary trees)** demonstrate how trees can be tailored for different storage and computational

needs. Additionally, **tree traversal techniques**, including inorder, preorder, postorder, and level-order, allow systematic visiting of nodes, each serving unique purposes like searching, sorting, and expression evaluation. Understanding these core concepts lays the foundation for more advanced applications in algorithms, data storage, and system design.

Recap

- ◆ A tree is a non-linear data structure made up of nodes connected in a hierarchical manner.
- ◆ Each tree has a unique root node, which is the starting point of the tree.
- ◆ A node contains data and pointers to its child nodes (left and right in binary trees).
- ◆ Binary trees are trees in which each node has at most two children—left and right.
- ◆ A full binary tree is a binary tree where every node has either 0 or 2 children.
- ◆ A complete binary tree is one where all levels are completely filled except possibly the last, and the last level is filled from left to right.
- ◆ An extended binary tree (or 2-tree) includes external (dummy) nodes so that every node has exactly 0 or 2 children.
- ◆ Trees are used in various applications such as file systems, databases, compilers, and AI decision-making.
- ◆ Tree traversal refers to visiting all the nodes of a tree in a specific order.
- ◆ Inorder traversal (Left → Root → Right) is mainly used in Binary Search Trees to retrieve data in sorted order.
- ◆ Preorder traversal (Root → Left → Right) is used to serialize or copy trees.
- ◆ Postorder traversal (Left → Right → Root) is useful in deleting trees or evaluating postfix expressions.
- ◆ Binary trees can be implemented recursively or using array/list representations, especially for complete binary trees.
- ◆ A strong understanding of binary trees and traversal methods is essential for learning advanced data structures and algorithms.

Objective Type Questions

1. What is a tree in data structures?
2. What is the root of a tree?
3. What is a leaf node in a tree?
4. What is a binary tree?
5. How many children can a node have in a binary tree?
6. What is a full binary tree?
7. What is a complete binary tree?
8. What is an extended binary tree or 2-tree?
9. What is a node in a tree structure?
10. What is meant by tree traversal?
11. Name the main types of tree traversal.
12. What is inorder traversal?
13. What is the order of preorder traversal?
14. What is the order of postorder traversal?
15. Which traversal gives sorted data in a Binary Search Tree?
16. What is the maximum number of nodes a binary tree node can have?
17. In which traversal is the root node visited first?
18. What is the use of postorder traversal?

Answers to Objective Type Questions

1. A tree is a non-linear data structure used to represent hierarchical data.
2. The root is the topmost node of the tree.
3. A leaf node is a node with no children.
4. A binary tree is a tree in which each node has at most two children.
5. Two children (left and right).

6. A full binary tree is one in which every node has either 0 or 2 children.
7. A complete binary tree has all levels completely filled except possibly the last, which is filled from left to right.
8. A 2-tree or extended binary tree includes external (dummy) nodes to make all nodes have 0 or 2 children.
9. A node is an element of a tree that contains data and links to child nodes.
10. Tree traversal is the process of visiting all the nodes of a tree in a specific order.
11. Inorder, Preorder, Postorder, and Level-order.
12. Inorder is left subtree \rightarrow root \rightarrow right subtree.
13. Preorder is root \rightarrow left subtree \rightarrow right subtree.
14. Postorder is left subtree \rightarrow right subtree \rightarrow root.
15. Inorder traversal.
16. Two.
17. Preorder traversal.
18. It is used for deleting a tree or evaluating expression trees.

Assignments

1. Explain the basic structure and terminology of a tree data structure. Discuss key components such as nodes, root, child, parent, siblings, and leaf nodes. Support your answer with a neat diagram and suitable examples.
2. Differentiate between Full Binary Tree, Complete Binary Tree, and Extended Binary Tree (2-tree). Define each type clearly and explain their structure, properties, and use cases. Include labeled diagrams for better understanding.
3. Describe the various types of tree traversal techniques. Explain inorder, preorder, and postorder traversal with proper algorithms, examples and proper explanations. Represent each traversal step with a binary tree diagram.

Reference

1. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). *Fundamentals of data structures in C* (2nd ed.). Silicon Press.
2. Thareja, R. (2014). *Data structures using C* (2nd ed.). Oxford University Press.
3. Weiss, M. A. (2007). *Data structures and algorithm analysis in C* (3rd ed.). Pearson Education.
4. Horowitz, E., & Sahni, S. (2007). *Data structures and algorithms in C* (2nd ed.). Universities Press.
5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Suggested Reading

1. GeeksforGeeks – Trees <https://www.geeksforgeeks.org/binary-tree-data-structure/>
2. TutorialsPoint – Data Structure Tree https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.html
3. Programiz – Tree Data Structure <https://www.programiz.com/dsa/tree>
4. JavaTpoint – Tree Data Structure <https://www.javatpoint.com/tree>
5. Khan Academy – Binary Trees <https://www.khanacademy.org/computing/computer-science/algorithms#binary-search-trees>

Unit 2

Binary Search Tree

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ define what a Binary Search Tree (BST) is
- ◆ list the basic operations of a BST
- ◆ identify the rules for node placement in a BST.
- ◆ familiarise the rules that determine the structure of a Binary Search Tree.

Prerequisites

Imagine you are managing a small personal library, and each book has a unique number on its spine. To keep the books organized and easy to find, you decide to place them in a way where every book with a smaller number goes to the left and every book with a larger number goes to the right. You start with one book at the center, and each time a new book arrives, you follow this rule to place it in the correct position. Over time, this forms a branching, tree-like structure that helps you quickly find, add, or remove books without rearranging the entire shelf. This method of organization is very similar to how a Binary Search Tree (BST) works, a data structure that stores elements in a sorted, efficient way, allowing for fast search, insertion, and deletion operations.

Key words

Node, Root, Successor, Leaf, Left Subtree, Right Subtree



Discussion

4.2.1 Introduction to Binary search Tree

A Binary Search Tree (BST) is a data structure commonly used in computer science to store and organize data in a sorted order. It follows all the characteristics of a binary tree, with the added condition that, for each node, all values in its left subtree are smaller, and all values in its right subtree are larger. This hierarchical arrangement as shown in Fig 4.2.1 enables efficient operations such as searching, inserting, and deleting data within the tree.

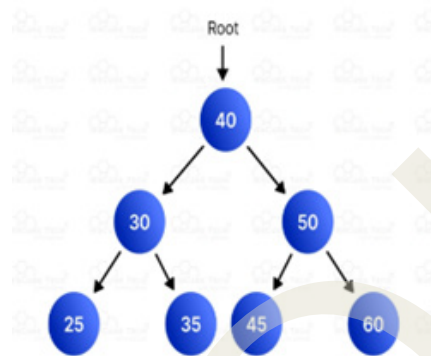


Fig 4.2.1 Binary Search Tree

The basic rule of a BST is that

1. The left subtree of a node contains only nodes with keys smaller than the key of that node.
2. The right subtree contains only nodes with keys greater than the key of that node.
3. Both left and right subtrees must themselves be valid binary search trees.
4. Duplicate keys are typically not allowed, though some BST variations handle duplicates in specific ways.

4.2.1.1 Binary Search Tree Properties

A Binary Search Tree has distinct characteristics that enable fast search, insertion, and deletion operations:

1. Node Components

Each node in a BST consists of the following:

- ◆ **Data:** The value stored in the node.
- ◆ **Left Child:** A reference or pointer to the node's left child.
- ◆ **Right Child:** A reference or pointer to the node's right child.

2. BinaryStructureTree

A BST is a specialized form of a binary tree, where each node can have up to two

children: one on the left and one on the right.

- ◆ **Node Ordering Rules**

- ◆ **Left Subtree:** All nodes in the left subtree hold values less than the node's value.
- ◆ **Right Subtree:** All nodes in the right subtree hold values greater than the node's value

Example:

Consider the following binary search tree in Fig 4.2.2

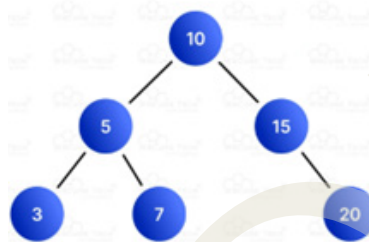


Fig 4.2.2 BST

- ◆ The root node has a value of 10.
- ◆ The left child of 10 is 5, and all nodes in the left subtree (3, 5, 7) have values less than 10.
- ◆ The right child of 10 is 15, and all nodes in the right subtree (15, 20) have values greater than 10.

4. No Repeated Values

A Binary Search Tree does not allow duplicate entries. Every value in the tree must be distinct to preserve the proper ordering.

5. Recursive Structure

Each subtree in a BST is itself a valid Binary Search Tree. This means both the left and right children of a node serve as roots of their own BSTs.

4.2.2 Operations in Binary Search Tree

The basic operations in a Binary Search Tree (BST) are

1. Search

Find whether a specific value exists in the tree.

2. Insertion

Add a new node to the tree while maintaining the BST property

Deletion

Remove a node from the tree and adjust the structure so the BST property still holds.

4.2.2.1 Searching in BST

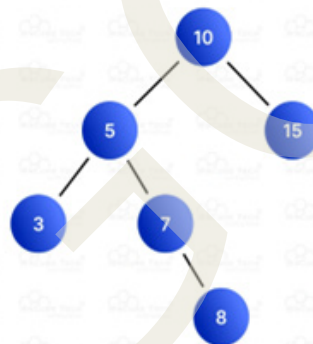
Searching in a binary search tree uses the BST property to efficiently locate a specific value by minimizing comparisons.

Algorithm:

1. Begin at the root node.
2. Compare the desired value with the current node's value.
3. If they are equal, the value has been found.
4. If the value is less, proceed to the left child.
5. If the value is greater, move to the right child.
6. Continue this process until the value is found or you reach a null node, indicating the value is not present in the tree.

Example:

Search for value 8 in the following BST



Step-by-Step Search:

- ◆ Begin at the root node (10).
- ◆ Since 8 is less than 10, go to the left child (5).
- ◆ 8 is greater than 5, so move to the right child (7).
- ◆ Again, 8 is greater than 7, so proceed to the right child (8).
- ◆ Now, 8 matches the current node's value, the search is successful.

4.2.2.2 Insertion in BST

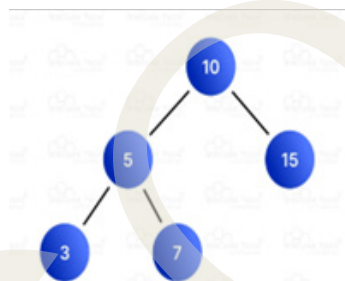
Inserting a new node into a Binary Search Tree involves placing it in the correct position to preserve the BST rule: all values in the left subtree must be less than the node, and all values in the right subtree must be greater.

Algorithm:

1. Start
2. If the tree is empty, insert the first element as the root node. All subsequent elements will be added as leaf nodes.
3. If a new element is smaller than the root, insert it as a leaf node in the left subtree.
4. If a new element is larger than the root, insert it as a leaf node in the right subtree.
5. Leaf nodes at the end do not have any child nodes and point to NULL.
6. End

Example:

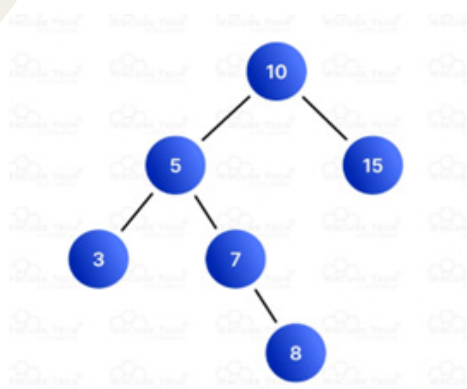
Insert the value 8 into the following BST



Step-by-Step Insertion:

- ◆ Begin at the root node (10).
- ◆ Since 8 is less than 10, go to the left child (5).
- ◆ Next, 8 is greater than 5, so move to the right child (7).
- ◆ Again, 8 is greater than 7, so proceed to the right child of 7, which is currently empty.
- ◆ Place 8 as the right child of node 7.

Result

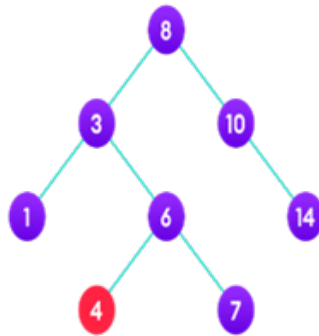


4.2.2.3 Deletion in BST

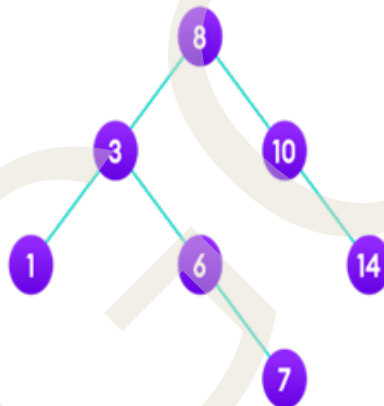
Deleting a node from a binary search tree requires adjusting the structure to preserve the BST rules. There are three cases to consider.

Case I: When the node to be removed is a leaf (i.e., it has no children), it can be deleted directly from the tree.

Example: Node 4 is to be deleted



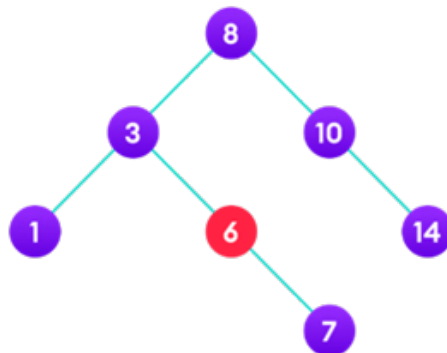
Delete the node



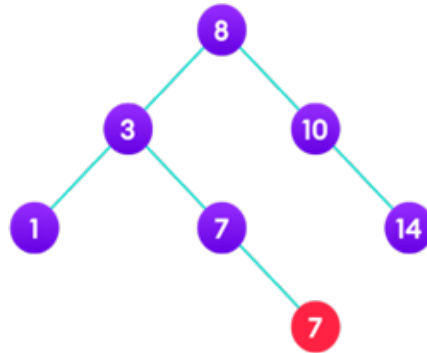
Case II: Node to be deleted has only one child

- ◆ Replace the node with its only child.
- ◆ Then, remove the child from its previous position in the tree.

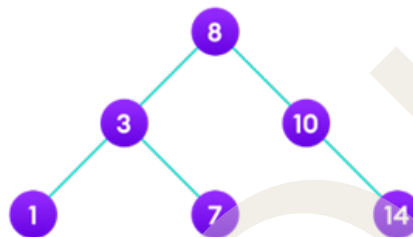
Example: Node 6 is to be deleted



Copy the value of its child to the node and delete the child



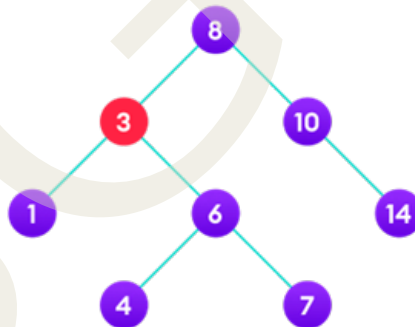
Final tree



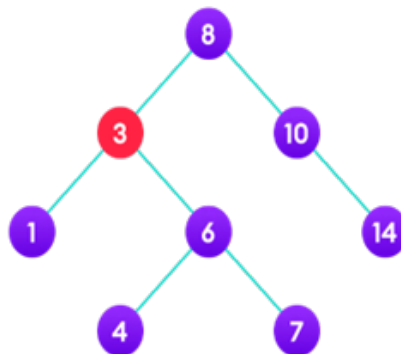
Case III: When the node to be deleted has two children, follow these steps:

- ◆ Find the inorder successor of the node.
- ◆ Replace the node's value with that of the inorder successor.
- ◆ Delete the inorder successor from its original location in the tree.

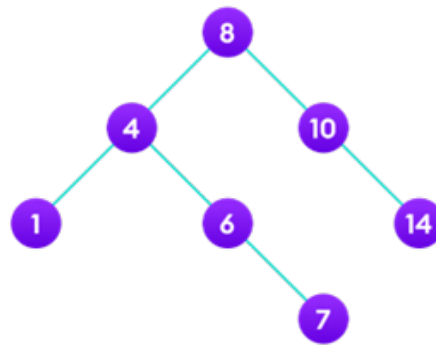
Example: Node 3 is to be deleted



Assign the value of the inorder successor (4) to the target node.



Delete the inorder successor



Recap

- ◆ A Binary Search Tree (BST) is a type of binary tree where data is stored in a sorted hierarchical order.
- ◆ In a BST:
 - The left subtree contains values smaller than the node.
 - The right subtree contains values greater than the node.
 - Both subtrees must also be valid BSTs.
 - Duplicate values are not allowed in a standard BST.
- ◆ Each node in a BST has:
 - A data values
 - A left child pointer/reference
 - A right child pointer/reference
- ◆ A node can have up to two children, forming a binary structure.
- ◆ BSTs are recursive, each subtree is itself a BST.
- ◆ Operations in BST:
 - **Searching:** Compares the target value with nodes from root to leaf.
 - **Insertion:** Places new nodes based on comparison (smaller to left, greater to right).
- ◆ Deletion: Three cases—
 - Node is a leaf: delete directly.
 - Node has one child: replace with the child.
 - Node has two children: replace with its inorder successor.

Objective Type Questions

1. What is the first node of a BST called?
2. How many children can a node in a BST have at most?
3. What kind of structure is a BST based on its children?
4. What is not allowed in a standard BST — duplicates or triplicates?
5. Which operation places a new node in its correct position?
6. Which operation removes a node from a BST?
7. Which node has no children in a BST?
8. Which node is used to replace a node with two children during deletion?
9. What do leaf nodes point to as children?
10. What type of search is used in BST?

Answers to Objective Type Questions

1. Root
2. Two
3. Binary
4. Duplicates
5. Insertion
6. Deletion
7. Leaf
8. Successor (or Inorder Successor)
9. NULL
10. Binary

Assignments

1. Explain the key properties of a Binary Search Tree (BST) and describe how these properties facilitate efficient search, insertion, and deletion operations.
2. Given the following sequence of values: 15, 10, 20, 8, 12, 16, 25, construct the corresponding Binary Search Tree. Illustrate the tree and explain the insertion process for each value.
3. Describe the three cases involved in deleting a node from a Binary Search Tree. Provide examples for each case to demonstrate how the deletion operation is performed while maintaining the BST properties.
4. Perform a search operation in the BST constructed from the sequence in question 2 to find the value 12. Detail each step taken during the search and explain why the BST structure makes the search efficient.
5. Discuss why duplicate values are typically not allowed in a BST. How might a BST variation handle duplicates? Propose a modification or approach to handle duplicates and explain its advantages and disadvantages.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson Education.
3. Sahni, S. (2005). *Data Structures, Algorithms, and Applications in C++* (2nd ed.). Universities Press.
4. Lafore, R. (2002). *Data Structures and Algorithms in Java* (2nd ed.). Sams Publishing.
5. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). *Fundamentals of Data Structures in C* (2nd ed.). Universities Press

Suggested Reading

1. Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). *Data Structures and Algorithms in C++* (2nd ed.). Wiley.
2. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
3. Malik, D. S. (2010). *Data Structures Using C++* (2nd ed.). Cengage Learning.
4. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.
5. McMillan, M. (2007). *Data Structures and Algorithms Using C#*. Cambridge University Press.

Unit 3

Balanced Binary Tree

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ explain the types of balanced trees such as AVL Tree, B-Tree, and B+ Tree.
- ◆ define the basic properties of AVL Tree, B-Tree, and B+ Tree.
- ◆ identify the main operations like insertion and deletion in AVL, B, and B+ Trees.
- ◆ describe where AVL, B, and B+ Trees are used in real-life applications.

Prerequisites

Before exploring balanced trees like AVL, B-Trees, and B+ Trees, learners should be familiar with the fundamentals of data structures. Concepts such as arrays, linked lists, stacks, and queues are the building blocks of efficient data handling. These structures help in organizing data in memory and are essential when moving on to trees. For example, queues are used in printer task management, while arrays are used in storing images. Understanding how these structures work prepares students to appreciate why trees are needed when data becomes large and complex.

Another important prerequisite is a basic understanding of binary trees and their variants. Learners should know what nodes, roots, children, leaves, height, and depth mean. Familiarity with binary search trees (BSTs) and traversal methods like inorder, preorder, and postorder is also necessary. These ideas form the foundation of balanced trees, which are designed to optimize performance. A real-life example of this is a contact list on your phone that quickly finds a name from thousands of entries; such efficiency is achieved through tree-based structures.

Finally, students should possess basic programming knowledge, preferably in languages like Python, C++, C or Java, to implement and experiment with tree operations such as insertion, deletion, and searching. Alongside coding skills, understanding algorithm efficiency (time and space complexity) is key. This helps in recognizing why balanced trees are used in real-world systems. For instance, B+ Trees are widely used in databases like MySQL and file systems like NTFS to manage and retrieve large data blocks swiftly, the same technology that helps retrieve a video from YouTube or a product from an e-commerce platform within seconds.



Key Concepts

AVL tree, B-tree, B+ tree, Balance factor, Height of a tree

Discussion

4.3.1 Introduction to Balanced Binary Tree

A **Balanced Binary Tree** is a special type of binary tree that maintains its structure in a way that keeps its height as small as possible. In a balanced tree, the difference in height between the left and right subtrees of any node is minimal, usually no more than one. This balance ensures that operations such as searching, insertion, and deletion remain efficient, typically running in logarithmic time. Balanced binary trees are essential in computer science because they prevent performance issues caused by skewed or uneven trees and are widely used in applications like databases, file systems, and memory management.

Figure 4.3.1 is an example for an unbalanced tree. Due to imbalances in the tree, there will be delay in transmission of information. In order to avoid such circumstances, the concept of balancing is introduced. A balanced tree is highly efficient when operations like insertion, deletion, etc. compared to unbalanced trees. In order for a tree to be perfectly height balanced, its left subtree and right subtree should be at the same level. The following figure 4.3.1 shows a perfectly balanced tree.

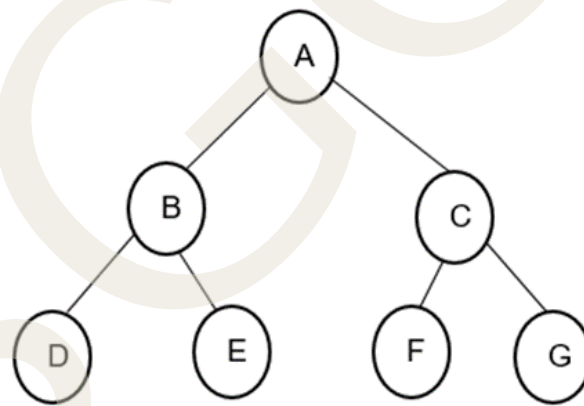


Fig 4.3.1 Perfectly balanced tree

In the above figure 4.3.1, the left subtree with root B and the right subtree with root C are at the same height. Hence, figure 4.3.1 is a perfectly balanced tree. However, perfectly balanced trees are rare. An alternative is 'almost' a perfectly balanced tree. A tree is said to be height-balanced if the heights of the left and right subtrees of each node are within 1. The following figure 4.3.2 shows an example of a height balanced tree that fits the definition of an 'almost' perfectly balanced tree.

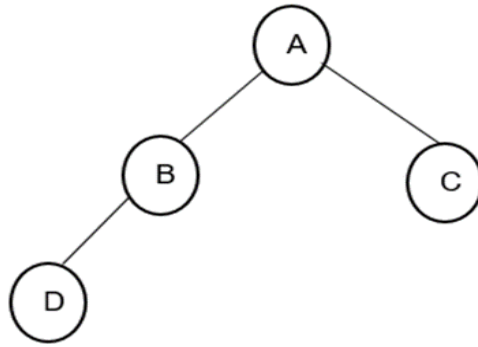


Fig 4.3.2 Height balanced tree

In the above figure 4.3.2, the difference between the left sub tree and the right subtree of each node is within the range 1. So the tree is called a height balanced tree.

4.3.1.1 Definition of Balanced Binary Tree

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1. Following are the conditions for a height-balanced binary tree (figure 4.3.3):

- ◆ difference between the left and the right subtree for any node is not more than one
- ◆ the left subtree is balanced
- ◆ the right subtree is balanced

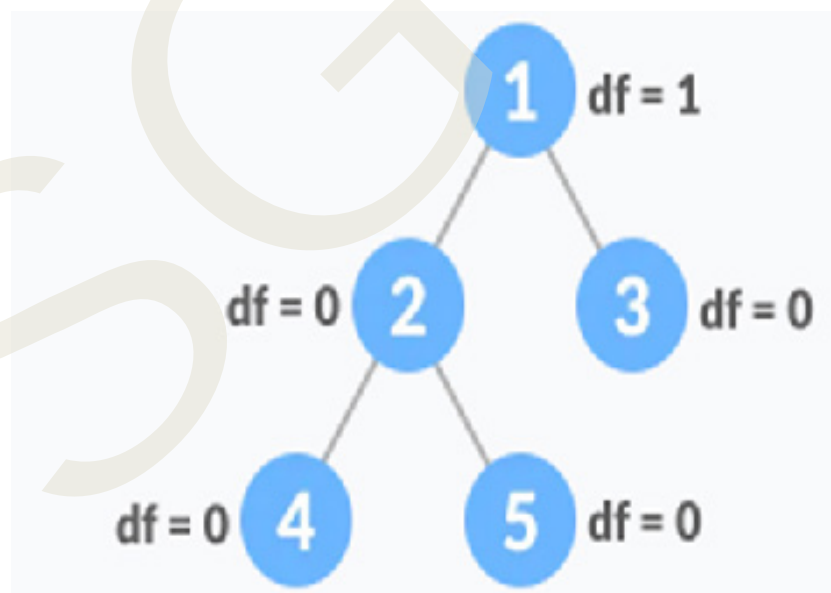


Fig 4.3.3 Balanced Binary Tree with depth at each level

The below figure 4.3.4 is an example for unbalanced binary tree. An unbalanced binary tree is a binary tree where the distribution of nodes among its left and right subtrees is uneven, leading to a significant difference in the heights of the subtrees at various

nodes. In extreme cases, an unbalanced binary search tree can degenerate into a structure resembling a linked list, where the height of the tree is close to the number of nodes ($O(n)$) rather than logarithmic ($O(\log n)$).

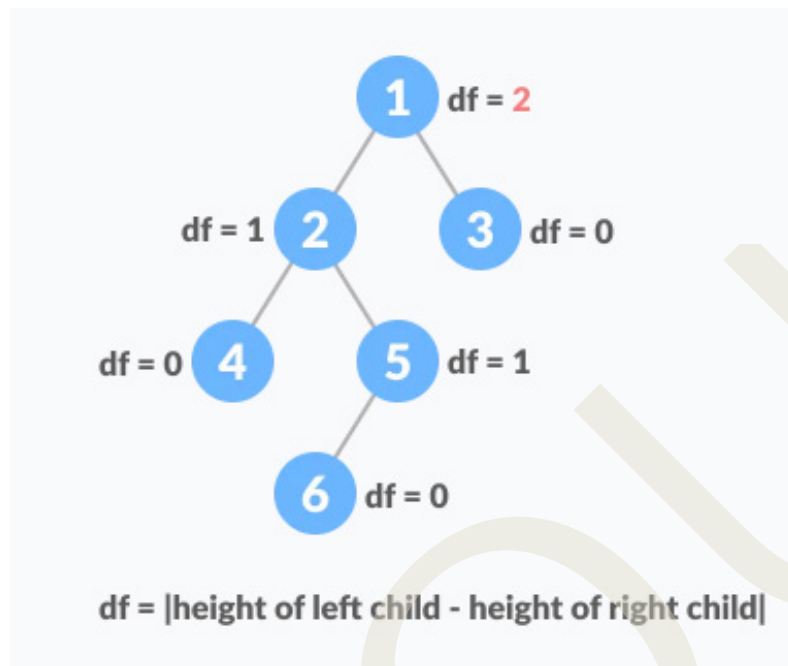


Fig 4.3.4 Unbalanced Binary Tree with depth at each level

4.3.1.2 Advantages of Balanced Binary Tree

Balancing a binary tree is essential for maintaining its efficiency and performance. When a binary tree remains balanced, it keeps its height minimal, ensuring that operations like searching, inserting, and deleting nodes can be performed quickly and effectively. Without balancing, the tree can become skewed, turning into a linear structure that slows down these operations. Balanced binary trees, such as AVL trees and Red-Black trees, are widely used in applications that require fast data access and manipulation. Understanding the advantages of tree balancing helps highlight its importance in optimizing both time and memory usage in computer programs.

1. **Faster Searching:** A balanced binary tree keeps its height low, ensuring that search operations take $O(\log n)$ time instead of $O(n)$, as in an unbalanced tree.
2. **Efficient Insertions and Deletions:** Operations like inserting or deleting nodes are faster in a balanced tree because fewer nodes need to be visited or adjusted.
3. **Prevents Skewed Structure:** Balancing avoids the formation of skewed trees (like linked lists), which can degrade performance and increase memory usage.
4. **Improved Performance in Large Data Sets:** Balanced trees handle large volumes of data more effectively, making them ideal for applications like databases, search engines, and file systems.

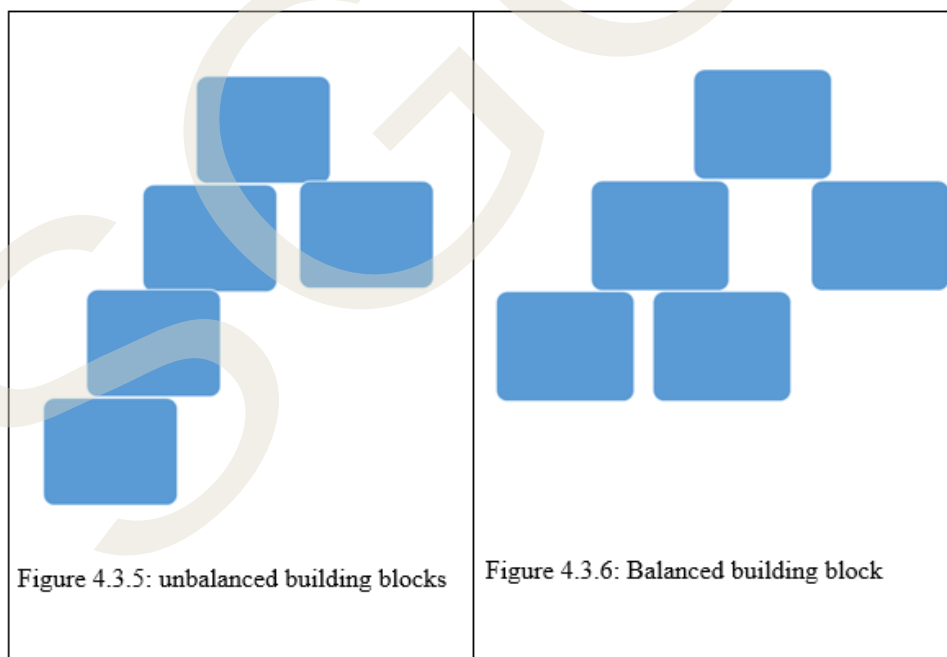
5. **Predictable Time Complexity:** Balancing ensures consistent **logarithmic time complexity**, which improves the reliability of programs that depend on tree operations.

Supports Real-Time Applications: Because of predictable and fast response times, balanced trees are useful in real-time systems such as **navigation, robotics, or mobile applications**.

4.3.2 AVL Tree

When the kids are playing with building blocks, when they want to make a model, as shown below, the chances of falling is more in figure 4.3.5 when compared to that of figure 4.3.6. In figure 4.3.5, the difference between the heights of blocks in the left side and right side is 2, while with figure 4.3.6, the difference between the heights of the blocks in the left side and right side is 1. Hence figure 4.3.6 is more stable than figure 4.3.5.

The above scenario can be compared with AVL trees. AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis who invented the AVL Tree in 1962. An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1. It is thus a balanced binary search tree. AVL trees are self-balancing, which means that the tree height is kept to a minimum so that a very fast runtime is guaranteed for searching, inserting and deleting nodes, with time complexity $O(\log n)$. AVL trees are particularly useful in applications where frequent lookups and updates are required while keeping the data structure efficient and balanced.



4.3.2.1 Properties of AVL Trees

AVL trees are a type of self-balancing binary search tree that maintain their structure through a strict balancing condition. Unlike regular binary search trees, which can become skewed and inefficient over time, AVL trees automatically adjust themselves

after each insertion or deletion to preserve balance. This ensures that the height of the tree remains logarithmic in relation to the number of nodes, allowing for consistently efficient performance in search, insert, and delete operations. The key properties of AVL trees make them highly suitable for applications where data access speed and structure balance are critical.

- ◆ An AVL tree is a self-balancing binary search tree where the height difference between the left and right subtrees of any node is at most one.
- ◆ Every node in an AVL tree maintains a balance factor calculated as the height of the left subtree minus the height of the right subtree, which must be -1, 0, or +1.
- ◆ The AVL tree maintains the binary search tree property, meaning the left child contains values less than the parent node and the right child contains values greater than the parent.
- ◆ AVL trees automatically perform rotations (single or double) during insertion and deletion operations to maintain balance.

4.3.2.2 Balance Factor

The **balance factor** is a key concept in AVL trees used to determine whether a node is balanced. It is defined as the **difference between the heights of the left and right subtrees** of a node.

$$\text{Balance Factor (BF)} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

Possible Values of Balance Factor:

- ◆ **+1** : Left subtree is one level taller than the right.
- ◆ **0** : Left and right subtrees are of equal height.
- ◆ **-1** : Right subtree is one level taller than the left.

If the balance factor of any node becomes **less than -1 or greater than +1**, the tree is considered **unbalanced**, and rotations (single or double) must be performed to restore balance.

For example: To find a balance factor, consider the below figure 4.3.7.

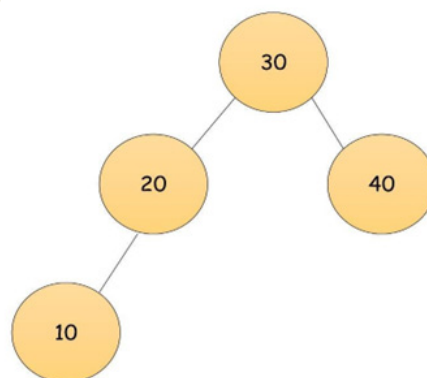


Fig 4.3.7 Example diagram to find balance factor

By the equation,

Balance Factor (BF) = Height(Left Subtree) - Height(Right Subtree)

- ◆ Balance Factor of node 40 = $0 - 0 = 0$ (Balanced)
- ◆ Balance Factor of node 30 = $2 - 1 = +1$ (Balanced)
- ◆ Balance Factor of node 20 = $1 - 0 = +1$ (Balanced)
- ◆ Balance Factor of node 10 = $0 - 0 = 0$ (Balanced)

4.3.2.3 Unbalanced Trees

As we discussed earlier, the tree becomes unbalanced whenever a node is inserted into a tree or whenever a node is deleted from a tree. When it is detected that the tree is unbalanced, we need to rebalance it. In the case of AVL trees, balancing is performed by **rotating nodes** either to the left or to the right. Following are categories of unbalanced trees:

- Case 1:** Left of left (LL) Unbalanced
- Case 2:** Right of right (RR) Unbalanced
- Case 3:** Right of left (RL) Unbalanced
- Case 4:** Left of right (LR) Unbalanced

Case 1: Left of left (LL) Unbalanced

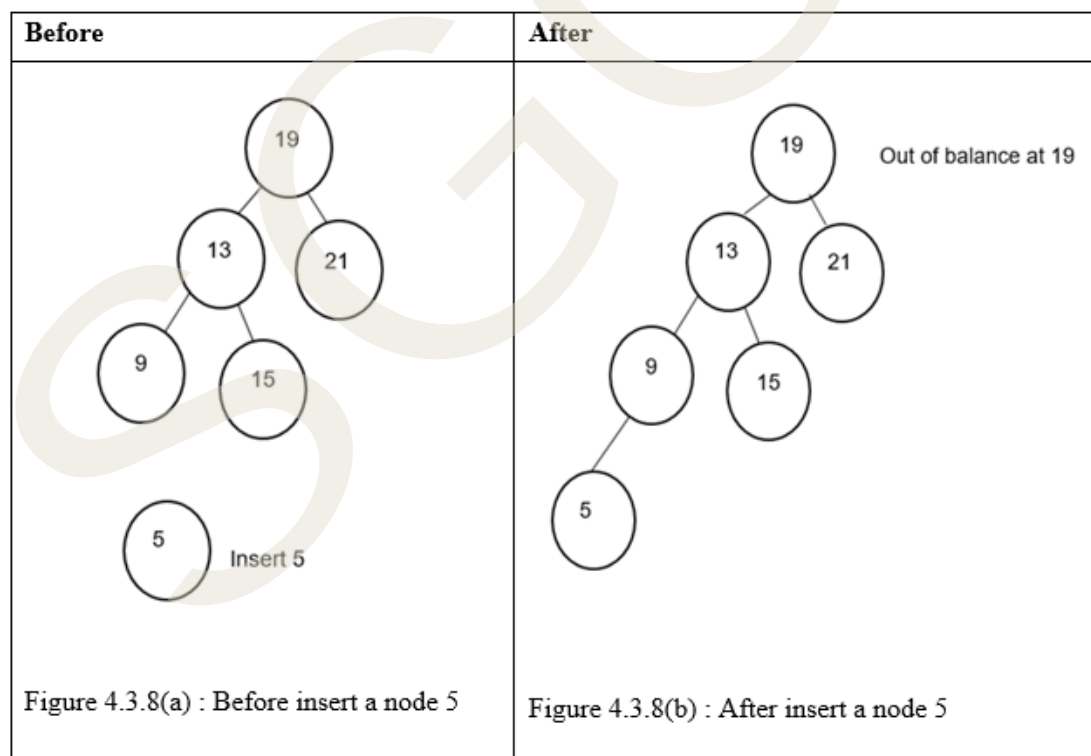
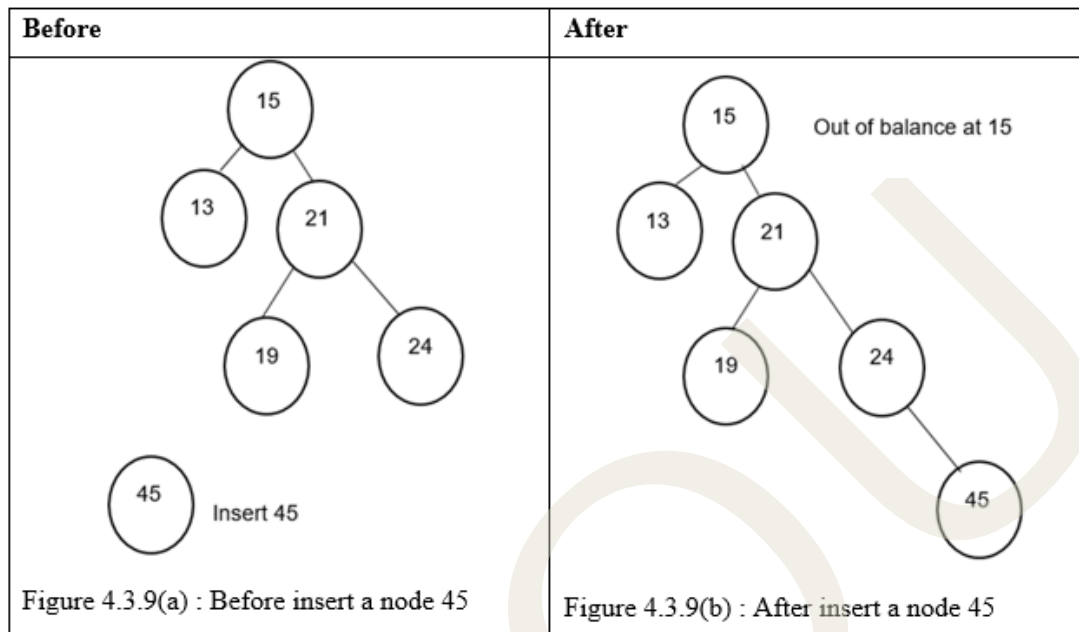


Figure 4.3.8(a) is a balanced tree as the difference between the height of the left subtree and the right subtree is 1. In the above figure 4.3.8(a), node 5 is to be inserted below node 9 resulting in figure 4.3.8(b). The figure 4.3.8(b) is out of balance at 19 as the dif-

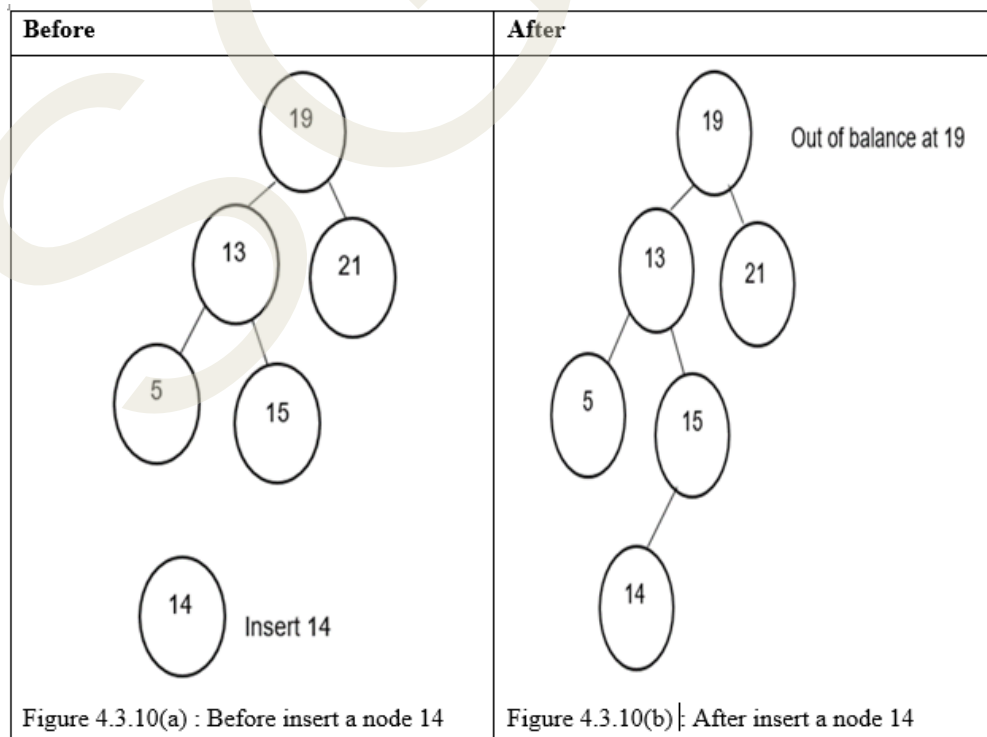
ference between the left sub tree and the right subtree is more than 1. Initially, Figure 4.3.14(a) is Left high. It is again left high after insertion of 9. Hence the name Left of Left.

Case 2: Right of right (RR) Unbalanced



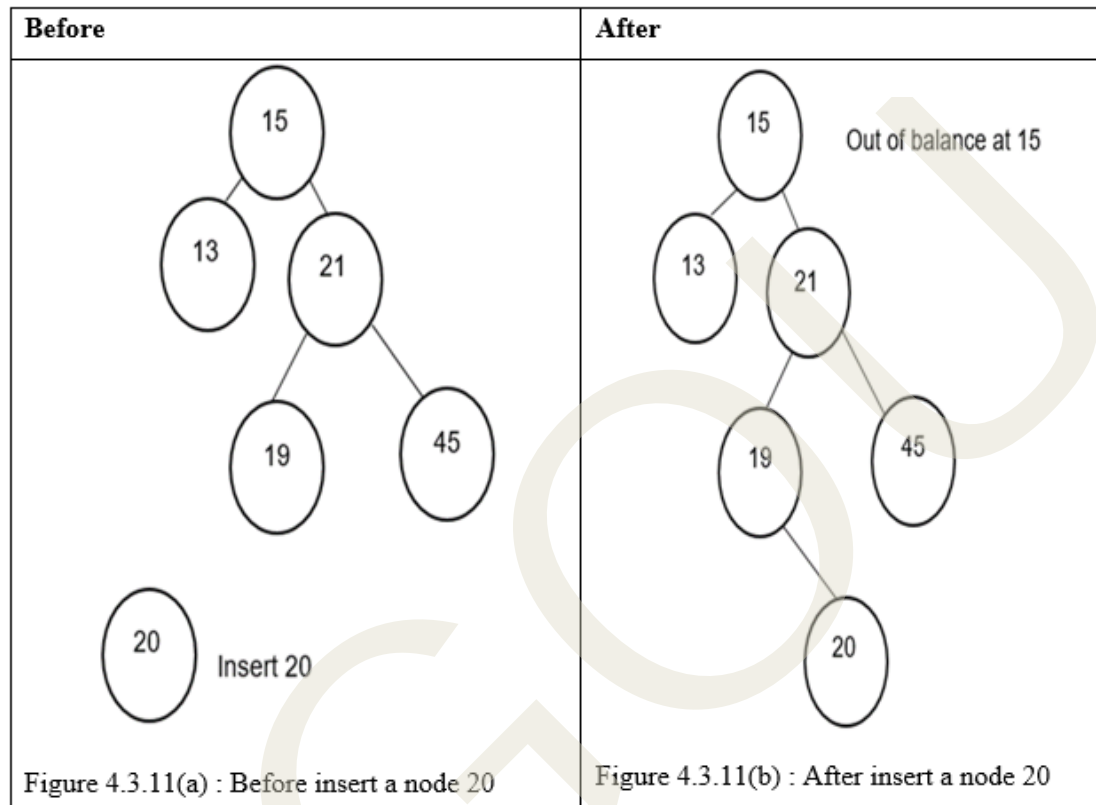
In the above figure 4.3.9(a), node 45 is to be inserted below node 24 resulting in figure 4.3.9(b). The figure 4.3.9(b) is out of balance at 15 as the difference between the left subtree and the right subtree is more than 1. Initially, Figure 4.3.9(a) is Right high. It is again Right high after insertion of 45. Hence the name Right of Right.

Case 3: Right of left (RL) Unbalanced



In the above figure 4.3.10(a), node 14 is to be inserted below node 15 as Left child resulting in figure 4.3.10(b). The figure 4.3.10(b) is out of balance at 19 as the difference between the left sub tree and the right subtree is more than 1. Initially, Figure 4.3.10(a) is left high. Here, a subtree that was left high has become right of left due to the insertion of 14. Hence the name Right of Left.

Case 4: Left of right (LR) Unbalanced



In the above figure 4.3.11(a), node 20 is to be inserted below node 19 as Right child resulting in figure 4.3.11(b). The figure 4.3.11(b) is out of balance at 15 as the difference between the left sub tree and the right subtree is more than 1. Initially, Figure 4.3.11(a) is right high. Here, a subtree that was right high has become left of right high due to the insertion of 20. Hence the name Left of Right.

Let us now discuss the ways to balance the out of balance trees with respect to case 1, case 2, case 3, and case 4.

The out of balance condition created by a left high subtree of a left high tree is balanced by rotating the out of balance node to the right. In the above figure 4.3.12(a), the node 21 tree is out of balance because the left subtree 19 is left high and it is on the left branch of node 21, which is also left high. In this case we balance the tree by rotating the root, 21, to the right so that it becomes the right subtree of 19.

Case 1: Left of Left Balancing

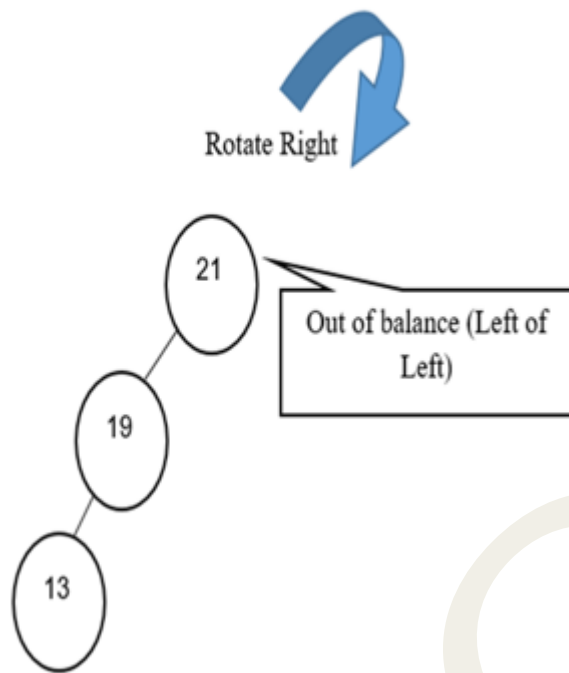


Figure 4.3.12(a) After inserting 13

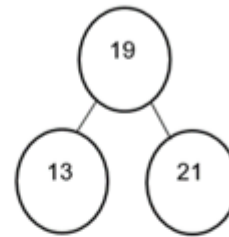


Figure 4.3.12(b) After rotation

Case 2: Right of Right Balancing

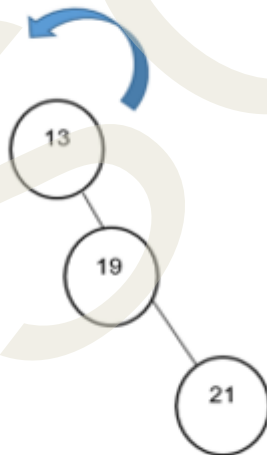


Figure 4.3.13(a) After inserting 21

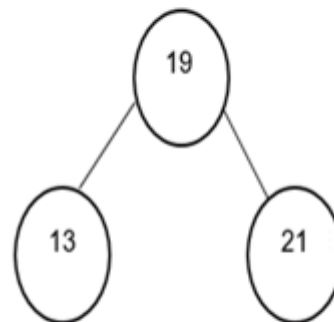


Figure 4.3.13(b) After rotation

Case 2 is an example for simple left rotation. Here, subtree 19 is balanced, but the root is not (Figure 4.3.13(a)). Hence, we rotate the root to the left, making it the left subtree of the new root, 19(Figure 4.3.13(b)).

Case 3: Right of Left Balancing

First rotate left

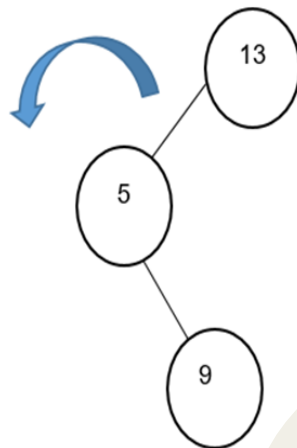


Figure 4.3.14(a) Left rotation

Second rotate right

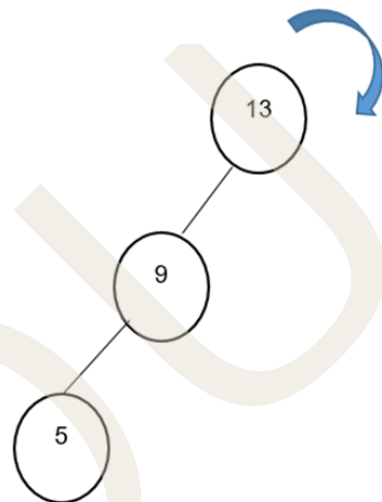


Figure 4.3.14(b) Right rotation

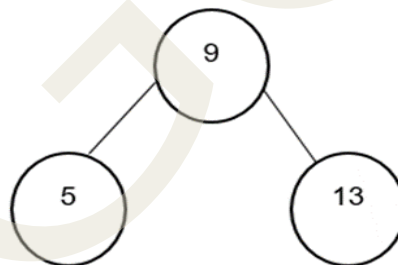


Figure 4.3.14(c) After balancing

We see an out-of-balance tree in which the root is left high and the left subtree is right high a right of left tree. To balance this tree, we first rotate the left subtree to the left (Figure 4.3.14(a)), then we rotate the root to the right (Figure 4.3.14(b)), making the left node the new root (Figure 4.3.14(c)).

Case 4: Left of Right Balancing

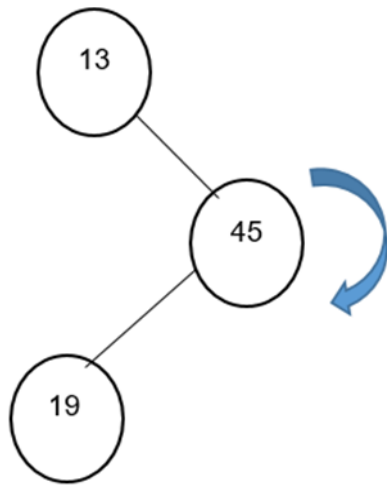


Figure 4.3.15(a) Right rotation

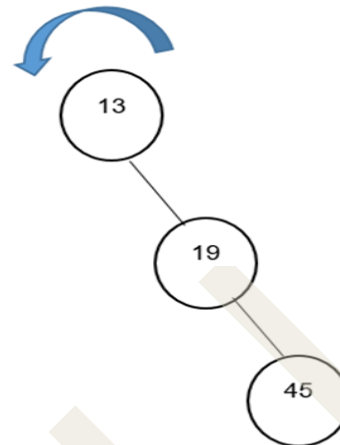


Figure 4.3.15(b) Left rotation

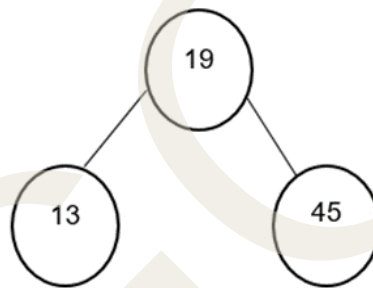


Figure 4.3.15(c) After balancing

To balance the tree, we first rotate the right subtree (45) right (Figure 4.3.15(a)) and then rotate the root (13) left (Figure 4.3.15(b)), resulting in Figure 4.3.15(c).

4.3.2.4 Rotations of AVL

In AVL trees, maintaining balance is crucial to ensure optimal performance in search, insertion, and deletion operations. Whenever an insertion or deletion causes the balance factor of any node to become greater than +1 or less than -1, the tree becomes unbalanced. To restore balance, **rotations** are performed. These rotations are specific tree restructuring operations that rearrange nodes without violating the binary search tree properties. Depending on the type and direction of imbalance, AVL trees use single or double rotations, making them a fundamental part of keeping the tree height-balanced. There are usually four cases of rotation in the balancing algorithm of AVL trees: **LL, RR, LR, RL**.

a. LL Rotation

LL rotation (Figure 4.3.16) is performed when the node is inserted into the right subtree

leading to an unbalanced tree. This is a single left rotation to make the tree balanced again.

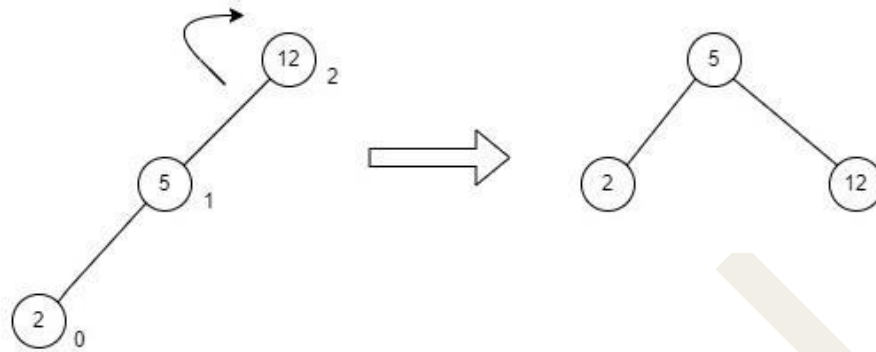


Fig 4.3.16 LL Rotation

The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node.

b. RR Rotation

RR rotation (Figure 4.3.17) is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again.

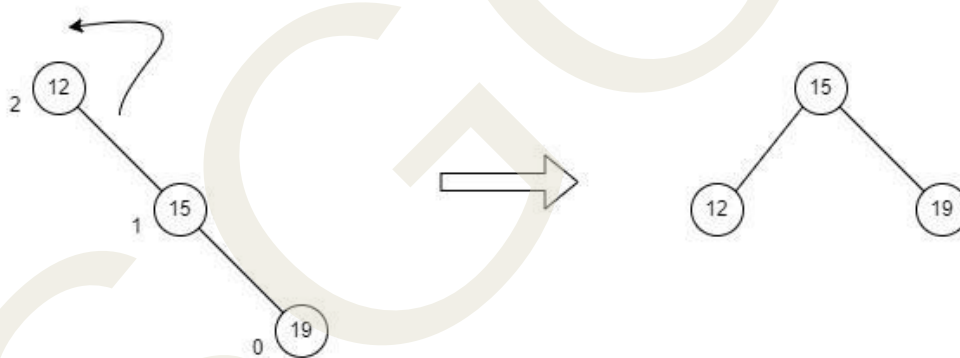


Fig 4.3.17 RR Rotation

The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node.

c. LR Rotation

LR rotation (Figure 4.3.18) is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. There are multiple steps to be followed to carry this out

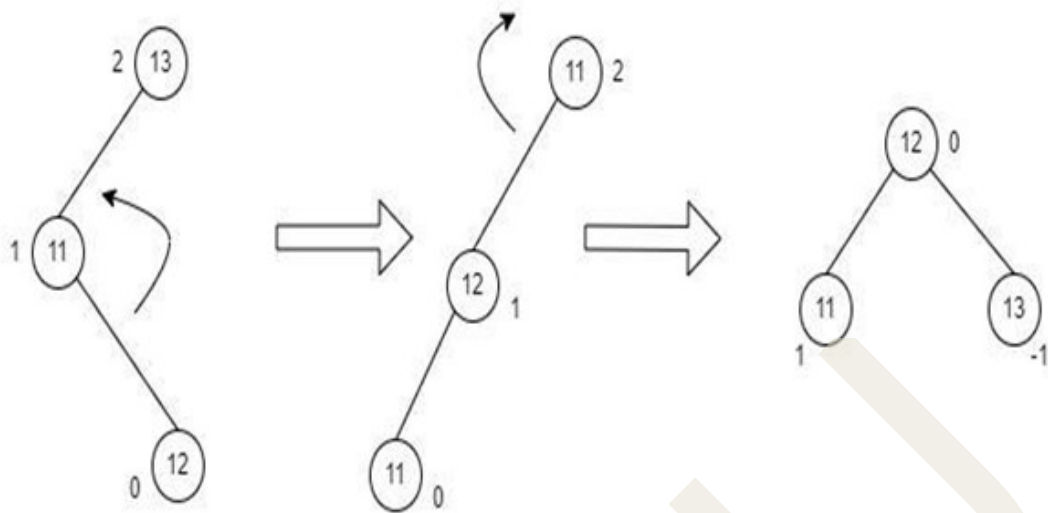


Fig 4.3.18 LR Rotation

d. RL Rotation

RL rotation (Figure 4.3.19) is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation. There are multiple steps to be followed to carry this out.

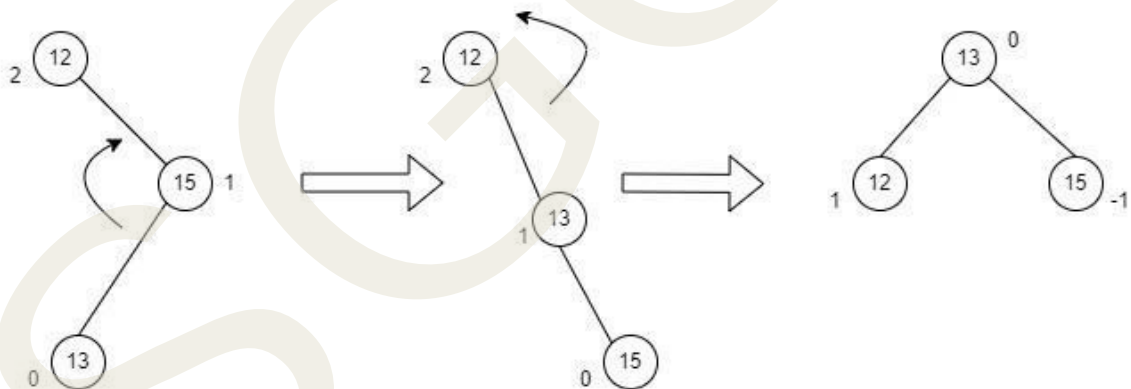


Fig 4.3.19 RL Rotation

4.3.2.5 Basic Operations of AVL Trees

The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are Insertion and Deletion.

a. Insertion operation

The data is inserted into the AVL Tree by following the Binary Search Tree property of

insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements. However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.

Algorithm: The following steps are involved in performing the insertion operation of an AVL Tree.

Step 1 : Create a node

Step 2 : Check if the tree is empty

Step 3 : If the tree is empty, the new node created will become the root node of the AVL Tree.

Step 4 : If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.

Step 5 : Suppose the balancing factor exceeds 1, we apply suitable rotations on the said node and resume the insertion from Step 4.

Let us understand the insertion operation by constructing an example AVL tree with 1 to 7 integers. Starting with the first element 1 (Figure 4.3.20), we create a node and measure the balance, i.e., 0.



Fig 4.3.20 Insert the node 1

Since both the binary search property and the balance factor are satisfied, we insert another element into the tree (Figure 4.3.21).

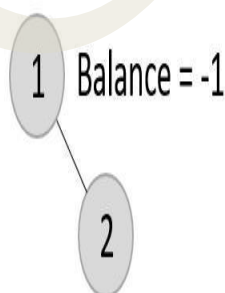


Fig 4.3.21 Insert the node 2 and find balance factor

The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree (Figure 4.3.22).

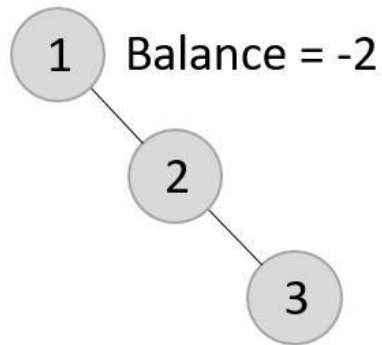


Fig 4.3.22 Insert the node 3 and find balance factor

Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes (Figure 4.3.23).

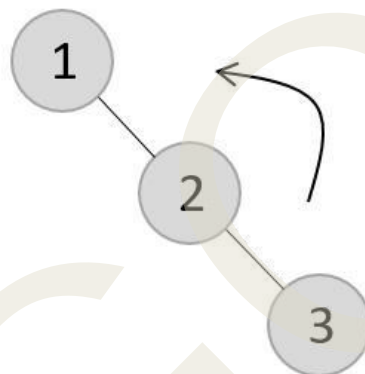


Fig 4.3.23 RR rotation

The tree is rearranged as shown in Figure 4.3.24.

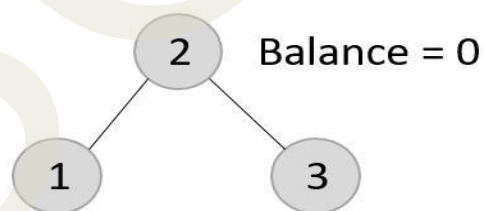


Fig 4.3.24 Rearranged nodes after RR rotation

Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as shown in Figure 4.3.25:

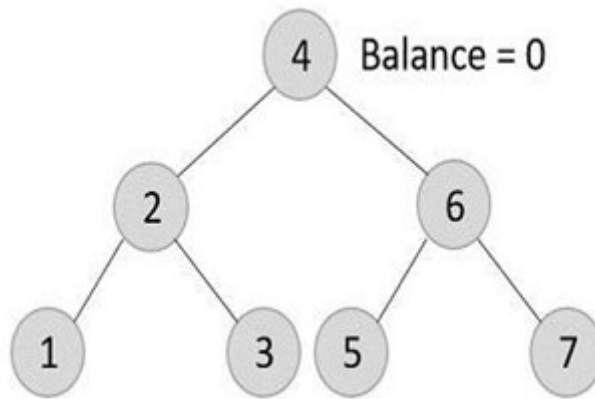


Fig 4.3.25 Final result

b. Deletion operation

Deletion in AVL trees involves removing a node while maintaining the tree's balanced structure. Similar to deletion in a standard binary search tree, the node is removed based on its position (leaf, one child, or two children). However, in an AVL tree, the deletion may disturb the height balance of the tree, causing some nodes to become unbalanced. To fix this, balance factors are updated as the algorithm moves up the tree, and rotations (single or double) are applied wherever necessary. This ensures that the AVL tree continues to provide efficient operations with $O(\log n)$ time complexity. Deletion in the AVL trees takes place in three different scenarios.

Scenario 1 (Deletion of a leaf node) : If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.

Scenario 2 (Deletion of a node with one child) : If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied.

Scenario 3 (Deletion of a node with two child nodes) : If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor value. Then try to delete the inorder successor node. If the balance factor exceeds 1 after deletion, apply balance algorithms.

Example: Using the same tree given above, let us perform deletion in three scenarios.

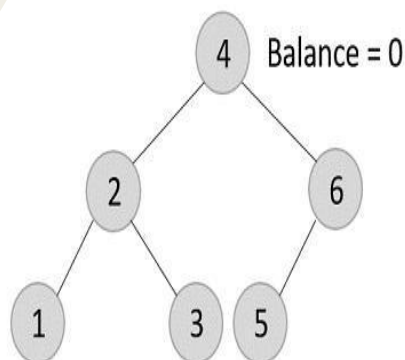


Fig 4.3.26 Example for deletion operation

Scenario 1: Deleting element 7 from the tree above figure 4.3.26. Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree figure 4.3.27.

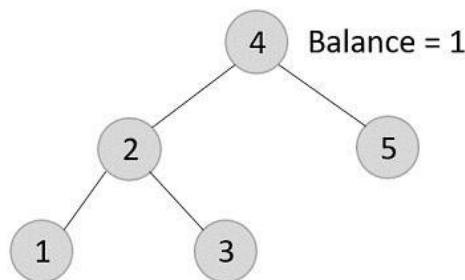


Fig 4.3.27 Result of scenario 1

Scenario 2: Deleting element 6 from the output tree achieved. However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node (node 5) figure 4.3.28.

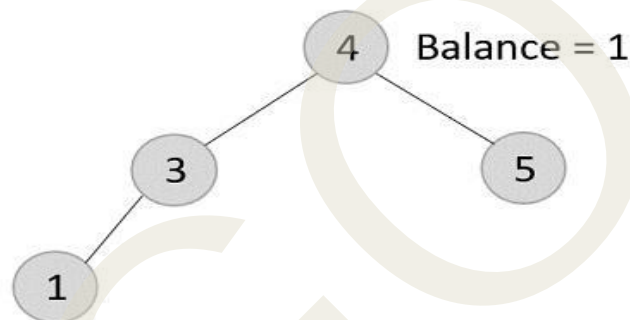


Fig 4.3.28 Scenario 2

The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4 figure 4.3.29.

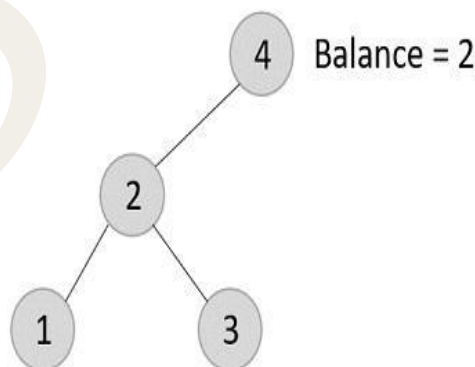


Fig 4.3.29 Apply rotations

The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here) figure 4.3.30.

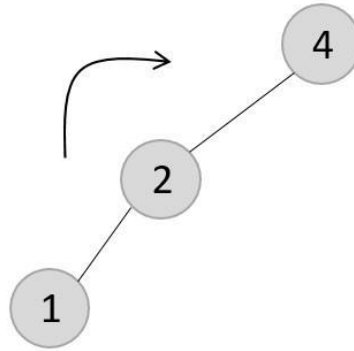


Fig 4.3.30 LL rotation

Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4 in figure 4.3.31.

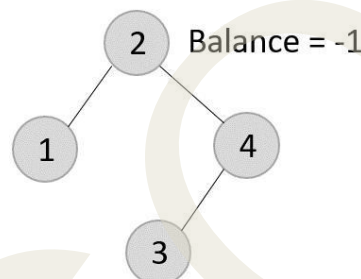


Fig 4.3.31 Final Result

Scenario 3: Deleting element 2 from the remaining tree. As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor in figure 4.3.32.

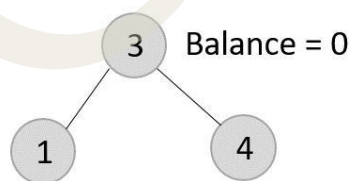


Fig 4.3.32 Final result of scenario 3

The balance of the tree still remains 1, therefore we leave the tree as it is without performing any rotations.

4.3.3.6 Applications of AVL Trees

AVL trees are widely used in areas where fast and efficient data access is critical, due to their self-balancing nature and guaranteed logarithmic time operations. By maintaining a balanced structure at all times, AVL trees ensure consistent performance for search, insert, and delete operations, even with large and dynamic datasets. Their ability to quickly adjust to changes makes them suitable for real-time applications, database

indexing, memory management, and any system where maintaining sorted data with optimal access speed is essential.

1. Database Indexing

AVL trees are used in database indexing systems to maintain balanced structures for fast data retrieval. Since search operations are $O(\log n)$, AVL trees provide consistent performance even with large datasets.

2. Memory Management

Operating systems and compilers use AVL trees for dynamic memory allocation. They help in tracking free and used memory blocks efficiently by maintaining a balanced tree of memory segments.

3. Search Engines

AVL trees can be used to implement search dictionaries or inverted indexes, allowing quick word lookups or phrase matching.

4. File Systems

Some file systems use AVL trees to organize files and directories for faster file access and directory traversal.

5. Routing Tables in Networks

AVL trees are used in networking for managing routing tables, where fast lookup of routes and paths is essential.

6. Gaming Applications

Games use AVL trees in decision-making logic (e.g., AI pathfinding) where frequent searches and updates are needed.

7. Symbol Tables in Compilers

Compilers use AVL trees to manage symbol tables, allowing quick insertions, deletions, and lookups of variables and function names.

8. Autocompletion and Spell Checkers

AVL trees help in storing dictionaries or word lists where fast prefix searching and retrieval is important.

9. Real-Time Systems

In real-time systems where guaranteed performance is required, AVL trees ensure operations complete in predictable time ($O(\log n)$).

4.3.4 B-Tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of

the binary search tree. B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of a huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is the height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, etc.

Following are the features of a B-Tree:

- ◆ All leaves are at the same level.
- ◆ A B-Tree is defined by the term minimum degree 't'. The value of t depends upon disk block size.
- ◆ Every node except root must contain at least $t-1$ keys. Root may contain a minimum 1 key.
- ◆ All nodes (including root) may contain at most $2t - 1$ keys.
- ◆ The number of children of a node is equal to the number of keys in it plus 1.
- ◆ All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in range from k_1 and k_2 .
- ◆ B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- ◆ Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\log n)$.

It is also known as a height-balanced m-way tree. The following Figure 4.3.33 shows a B tree. Here, the root node contains more than 1 key value (Here, we have 3 pointers and 2 key values). Also, the root node has more than two children.

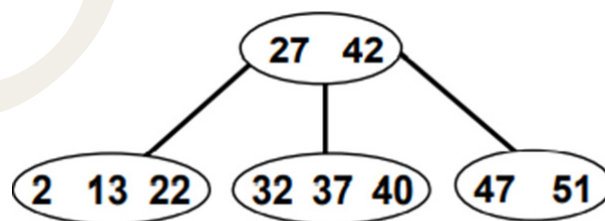


Fig 4.3.33 B –Tree

The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity.

4.3.4.1 Operations on B-Tree

B-Trees support various operations that make them highly efficient for managing large datasets. Below are the key operations in table 4.3.1:

Table 4.3.1 basic key operations and time complexity of B-Tree

Sl. No.	Operation	Time Complexity
1.	Search	$O(\log n)$
2.	Insert	$O(\log n)$
3.	Delete	$O(\log n)$

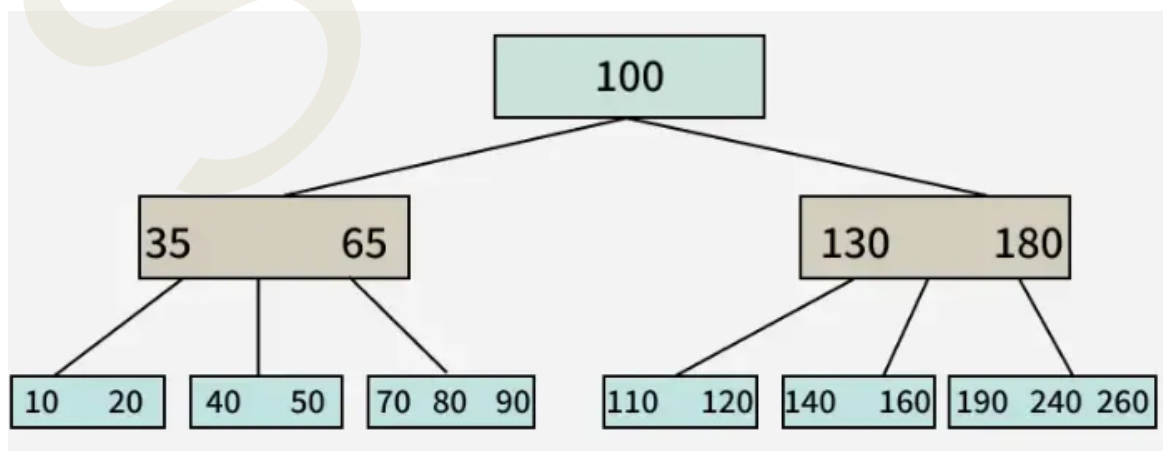
a. Search Operation in B-Tree

Search is similar to the search in Binary Search Tree. Let the key to be searched is k .

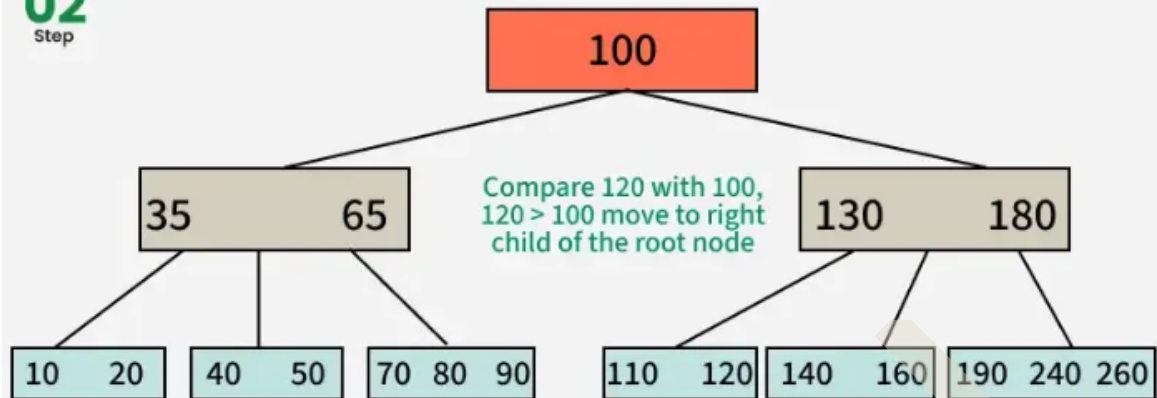
- ◆ Start from the root and recursively traverse down.
- ◆ For every visited non-leaf node
 - If the current node contains k , return the node.
 - Otherwise, determine the appropriate child to traverse. This is the child just before the first key greater than k .
- ◆ If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching for a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

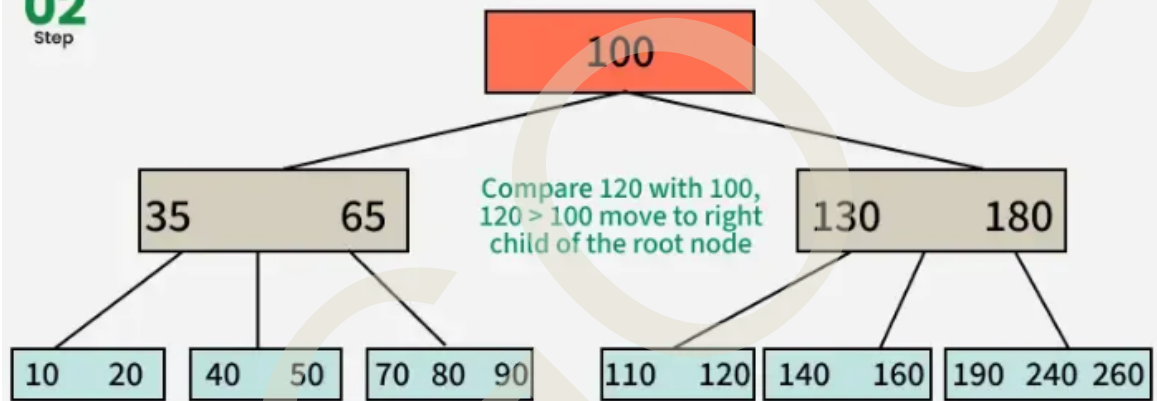
Example: Search 120 in the given B-Tree in figure 4.3.34.



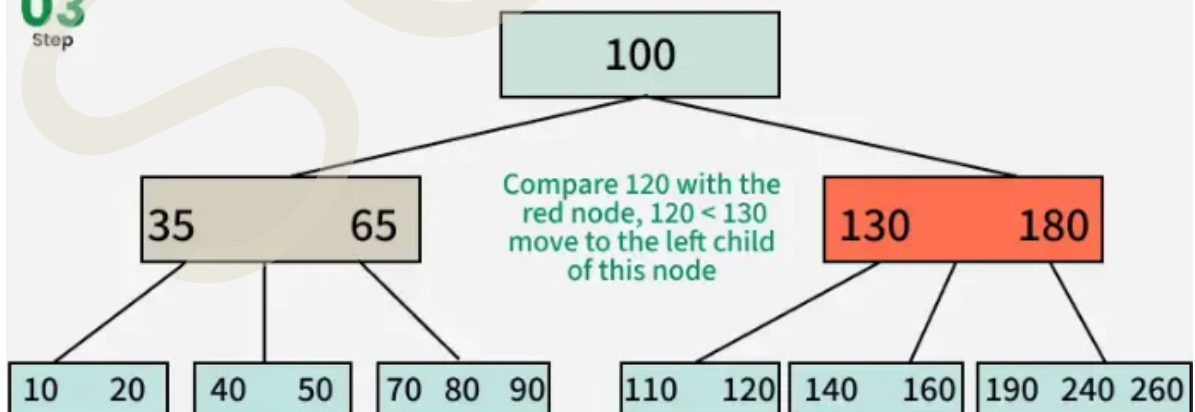
02
Step



02
Step



03
Step



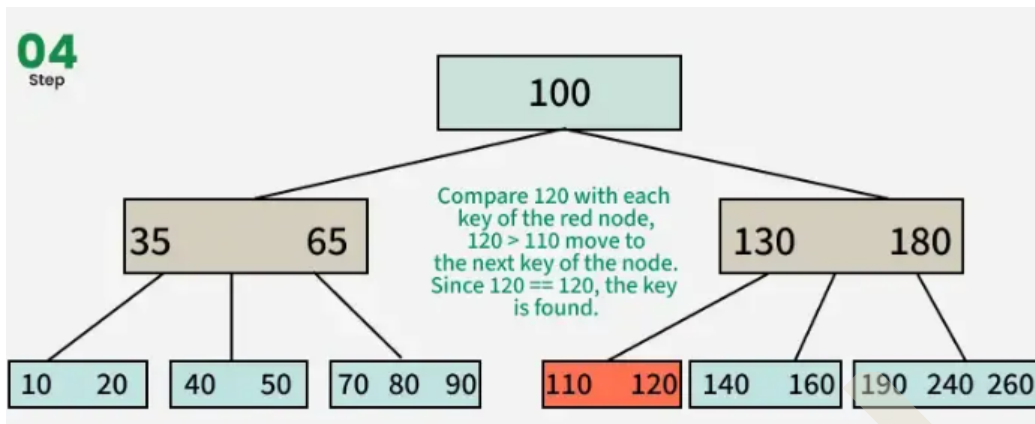


Fig 4.3.34 Search operation of B –Tree

The key 120 is located in the leaf node containing 110 and 120. The search process is complete.

b. Insertion operation

The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure.

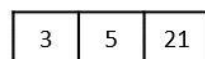
Step1 :Calculate the maximum ($m-1$) and, minimum ($\lceil \frac{m}{2} \rceil - 1$) number of keys a node can hold, where m is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys ($m - 1$) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil - 1$) = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

Step 2 :The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children in figure 4.3.35.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



Adding 9 will cause overflow in the node; hence it must be split.

Fig 4.3.35 Inserting operation

Step 3 :All the leaf nodes must be on the same level in figure 4.3.36.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

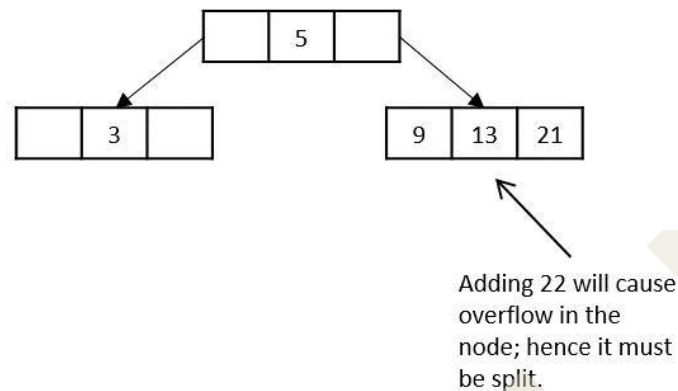


Fig 4.3.36 Overflow occur in insert operation

Step 4 : The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued in figure 4.3.37.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

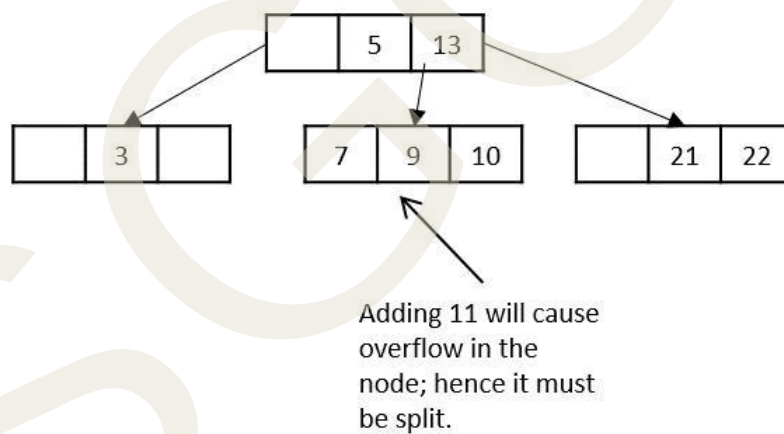


Fig 4.3.37 Overflow occur while adding 11

Another problem occurs during the insertion of 11, so the node is split and the median is shifted to the parent.

While inserting 16, even if the node is split in two parts, the parent node also overflows (figure 4.3.38) as it reaches the maximum keys. Hence, the parent node is split first, and the median key becomes the root. Then, the leaf node is split in half and the median of the leaf node is shifted to its parent.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

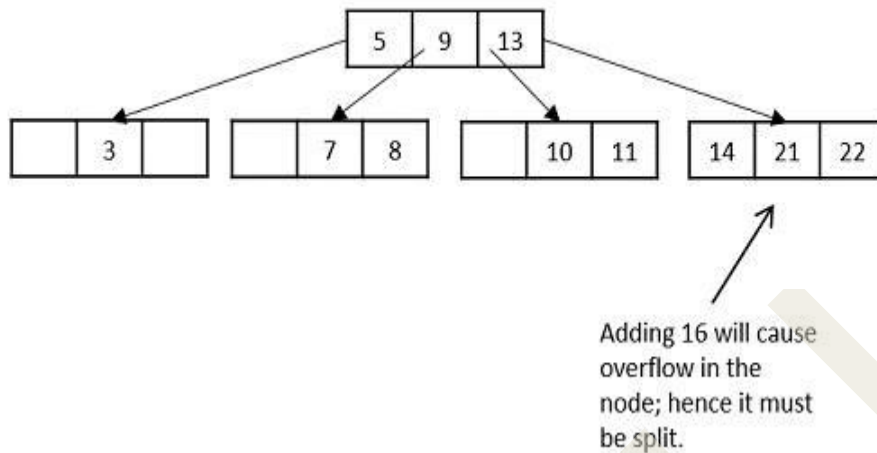


Fig 4.3.38 Overflow occur in insertion of 16

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

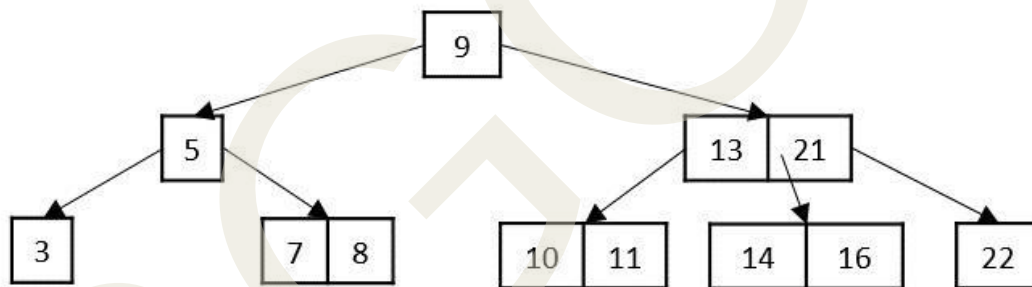


Fig 4.3.39 Final Result

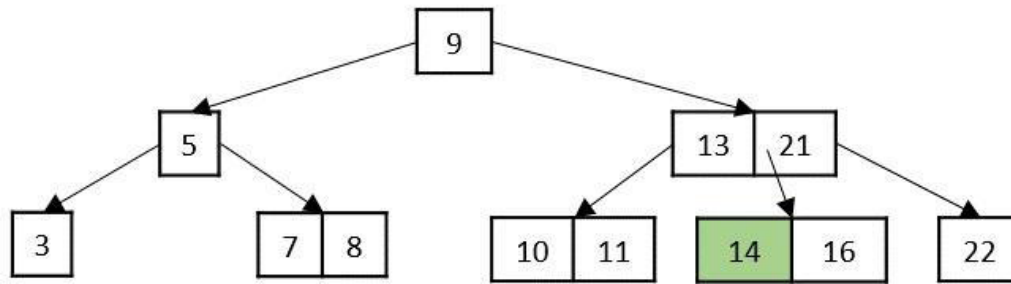
The final B tree after inserting all the elements is achieved in figure 4.3.39.

c. Deletion operation

The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows:

Case 1 : If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node in figure 4.3.40.

delete key 14



delete key 14

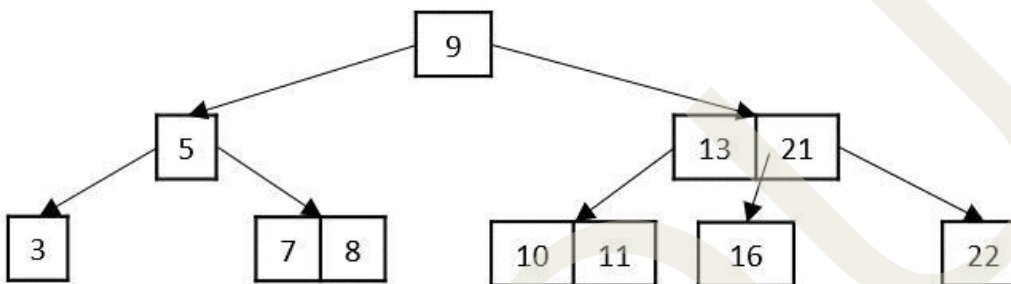
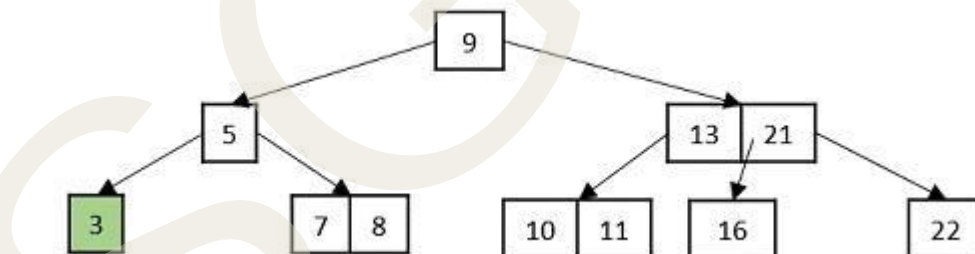


Fig 4.3.40 Delete key 14

Case 2 : If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them in figure 4.3.41.

delete key 3



delete key 3

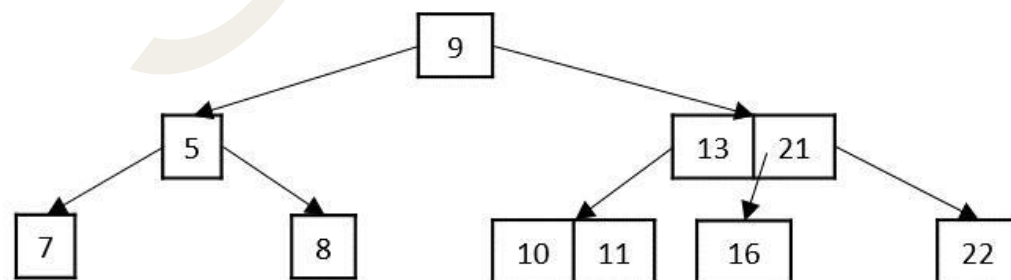
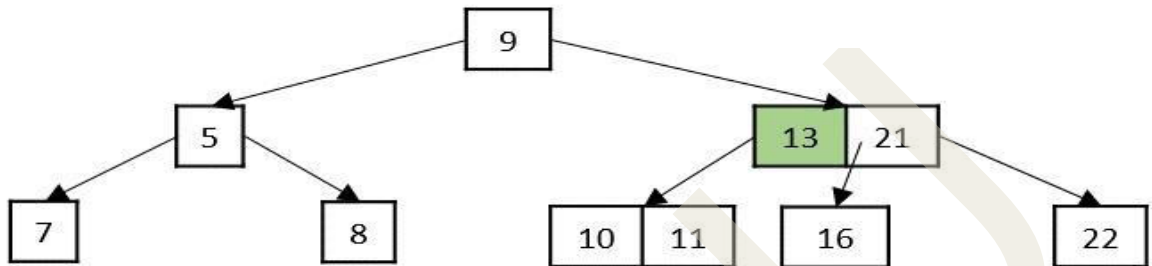


Fig 4.3.41 Delete key 3

Case 3 : If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have a minimum number of keys, they're merged together in figure 4.3.42.

Delete key 13



Delete key 13

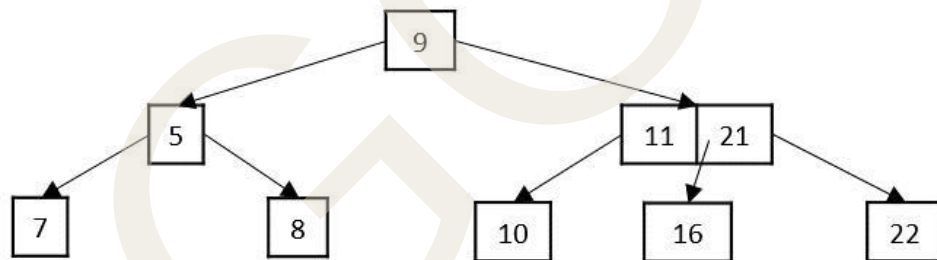
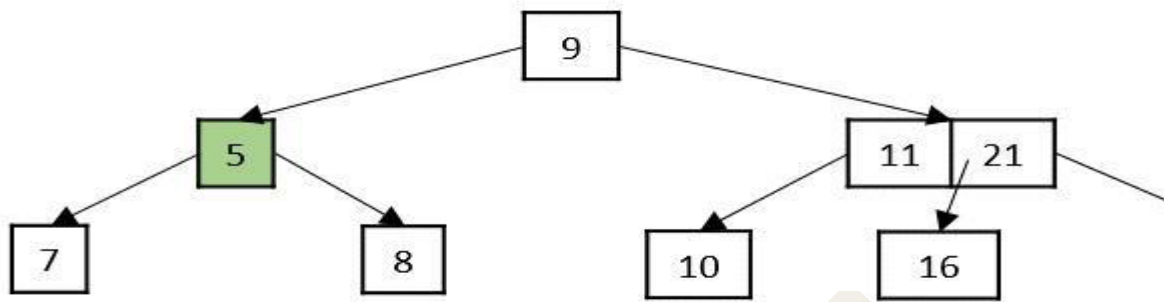


Fig 4.3.42 Delete key 13

Case 4 : If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent in figure 4.3.43.

Delete key 5



Delete key 5

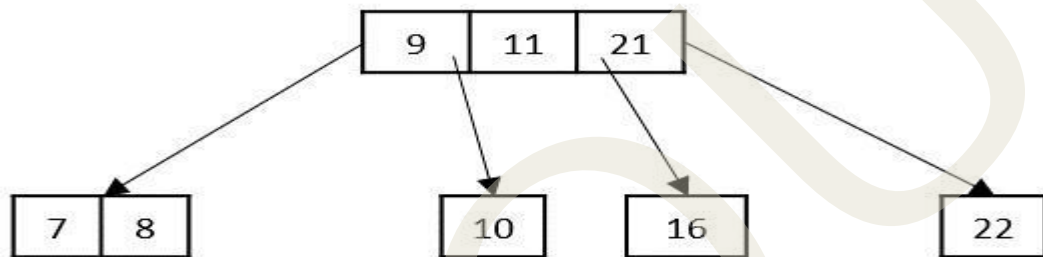


Fig 4.3.43 Delete key 5

4.3.4.2 Applications of B-Trees

B-Trees are powerful data structures that provide efficient and balanced storage for large volumes of sorted data. Due to their ability to minimize disk access and maintain sorted order, B-Trees are widely used in systems that handle extensive read/write operations. Their structure allows fast search, insertion, and deletion, even with millions of records, making them ideal for use in databases, file systems, and indexing mechanisms. As a result, B-Trees have become a fundamental component in both traditional and modern computing environments where performance and scalability are essential.

- ◆ It is used in large databases to access data stored on the disk
- ◆ Searching for data in a data set can be achieved in significantly less time using the B-Tree
- ◆ With the indexing feature, multilevel indexing can be achieved.
- ◆ Most of the servers also use the B-tree approach.
- ◆ B-Trees are used in CAD systems to organize and search geometric data.
- ◆ B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

4.3.4.3 Advantages and Disadvantages of B-Trees

B-Trees are widely used in computer science for managing and organizing large sets of sorted data efficiently. Their balanced structure ensures that operations such as search, insert, and delete are performed in logarithmic time, making them suitable for systems that rely on fast data access and minimal disk reads. However, like any data structure, B-Trees have their own strengths and weaknesses. Understanding the advantages and disadvantages of B-Trees is essential for determining their suitability in various applications such as databases, file systems, and indexing systems.

Some of the advantages of B-Trees are:

- ◆ B-Trees have a guaranteed time complexity of $O(\log n)$ for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- ◆ B-Trees are self-balancing.
- ◆ High-concurrency and high-throughput.
- ◆ Efficient storage utilization.

Some of the disadvantages of B-Trees are:

- ◆ B-Trees are based on disk-based data structures and can have a high disk usage.
- ◆ Not the best for all cases.
- ◆ For small datasets, the search time in a B-Tree might be slower compared to a binary search tree, as each node may contain multiple keys.

In summary, B-Trees are a smart and efficient way to handle large amounts of data. Their balanced structure and ability to store multiple keys in one node make searching, adding, and deleting data fast and reliable. B-Trees are especially useful for systems like databases and file storage, where quick access to data is important.

4.3.5 B+ Tree

The B+ trees (figure 4.3.44) are extensions of B trees designed to make the insertion, deletion and searching operations more efficient. The properties of B+ trees are similar to the properties of B trees, except that the B trees can store keys and records in all internal nodes and leaf nodes while B+ trees store records in leaf nodes and keys in internal nodes. One profound property of the B+ tree is that all the leaf nodes are connected to each other in a single linked list format and a data pointer is available to point to the data present in the disk file. This helps fetch the records in equal numbers of disk access.

Since the size of main memory is limited, B+ trees act as the data storage for the records that couldn't be stored in the main memory. For this, the internal nodes are stored in the main memory and the leaf nodes are stored in the secondary memory storage.

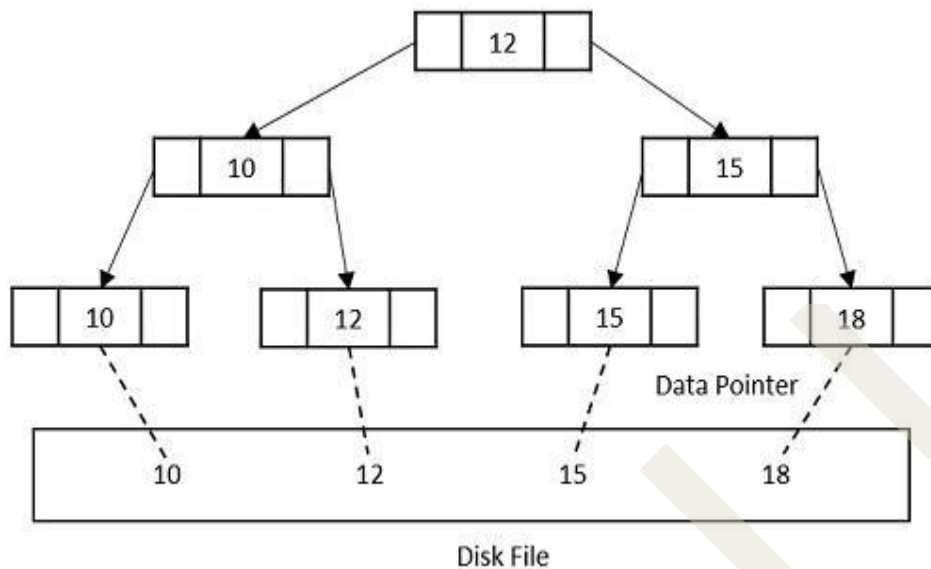


Fig 4.3.44 Structure of B+ Tree

Properties of B+ trees

Every node in a B+ Tree, except root, will hold a maximum of m children and $(m-1)$ keys, and a minimum of $\lceil m/2 \rceil$ children and $\lceil m/2 \rceil$ keys, since the order of the tree is m .

The root node must have no less than two children and at least one search key.

All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.

A B+ tree always maintains sorted data.

4.3.5.1 Basic Operations of B+ Trees

The operations supported in B+ trees are Insertion, deletion and searching with the time complexity of $O(\log n)$ for every operation. They are almost similar to the B tree operations as the base idea to store data in both data structures is the same. However, the difference occurs as the data is stored only in the leaf nodes of a B+ trees, unlike B trees.

a. Insertion operation

The insertion to a B+ tree starts at a leaf node.

Step 1 : Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4

Maximum Children $(m) = 4$

Minimum Children $\left(\left\lceil \frac{m}{2} \right\rceil\right) = 2$

Maximum Keys $(m - 1) = 3$

Minimum Keys $\left(\left\lceil \frac{m-1}{2} \right\rceil\right) = 1$

Step 2 : Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number in figure 4.3.45.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

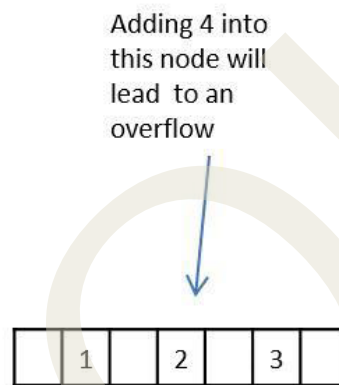


Fig 4.3.45 Insertion of B+ Tree

Step 3 : The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child in figure 4.3.46.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

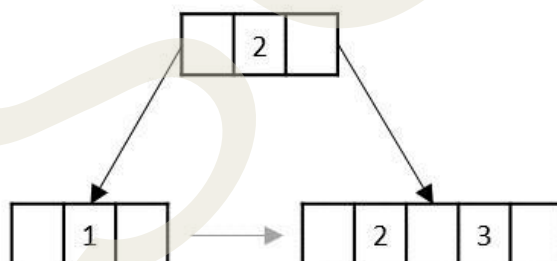


Fig 4.3.46 Insertion of nodes

Step 4 : But if the internal node also exceeds the maximum key property, the node is split in half where the left child consists of the minimum keys and remaining keys are stored in the right child. However, the smallest number in the right child is made the parent in figure 4.3.47.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

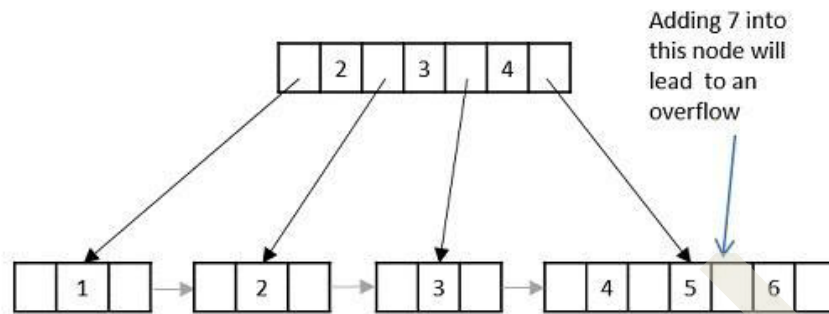


Fig 4.3.47 Insertion of node 7

Step 5 : If both the leaf node and internal node have the maximum keys, both of them are split in the similar manner and the smallest key in the right child is added to the parent node in figure 4.3.48.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

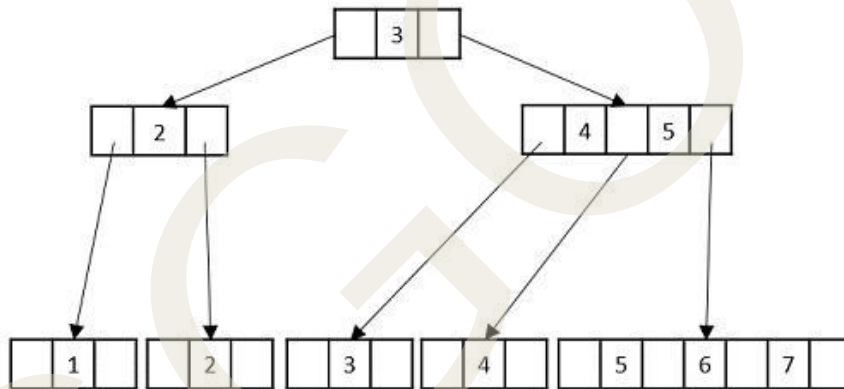


Fig 4.3.48 Final result

4.3.5.2 Applications of B+ Trees

B+ Trees are widely used in systems that manage large and sorted datasets, thanks to their efficient and scalable structure. By storing all data values at the leaf level and maintaining a linked list of leaves, B+ Trees support fast and smooth traversal for both point queries and range queries. This makes them ideal for use in databases, file systems, and search engines where high-performance data access is required.

- 3. Database Indexing:** B+ Trees are extensively used in relational databases (e.g., MySQL, Oracle) to organize and access large datasets efficiently.
- 4. File Systems:** Many modern file systems (e.g., NTFS, ReiserFS) use B+ Trees to manage directories and metadata due to their fast sequential and random access.

5. **Data Warehousing and OLAP Systems:** B+ Trees support range-based queries and ordered traversals, making them ideal for analytical workloads.
6. **Search Engines:** Used in inverted indexes for keyword searches where large volumes of data need quick lookup.
7. **Operating Systems:** B+ Trees are used in memory and storage management for indexing and accessing data blocks.
8. **Key-Value Stores:** Many NoSQL databases and distributed storage systems use B+ Trees to handle sorted keys and efficient access.

4.3.5.3 Advantages of B+ Trees

The design of B+ Trees offers several advantages over traditional tree structures. Their shallow height and uniform data access from leaf nodes ensure minimal disk I/O and consistent performance. The linked structure of leaf nodes also enables fast sequential access, making B+ Trees especially effective for range queries and full-table scans. These strengths make B+ Trees a preferred choice in data-intensive applications.

1. **Efficient Range Queries:** All data is stored in linked leaf nodes, allowing fast and easy traversal for range-based and sequential queries.
2. **Better Disk Access:** Internal nodes store only keys (no data), making nodes smaller and reducing the number of disk accesses.
3. **All Values at Leaf Level:** Uniform access time for all records since data is retrieved only from the leaf level.
4. **Linked Leaves:** The leaf nodes are linked as a linked list, which makes full table scans and ordered access very efficient.
5. **Shallower Trees:** Compared to B-Trees, B+ Trees are often shallower, meaning fewer I/O operations are needed.

4.3.5.4 Disadvantages of B+ Trees

Despite their benefits, B+ Trees also have some drawbacks. Because data is only stored at the leaf level, even exact-match queries must traverse to the bottom of the tree, which can slightly increase access time. Additionally, maintaining internal index nodes and the linked list of leaves adds implementation complexity and memory overhead. These trade-offs should be considered when deciding to use B+ Trees in a system.

1. **Slower Point Queries:** Since all data is stored only in the leaves, even exact-match queries must reach the leaf level.
2. **More Storage for Internal Nodes:** Additional space is needed for internal nodes to maintain multiple keys and pointers.
3. **Complex Implementation:** The logic for insertion, deletion, and re-balancing is more complex compared to simpler trees.

Overhead of Leaf Links: Maintaining the linked structure of leaves introduces additional memory and update overhead.

Balanced binary trees are essential data structures that maintain efficient performance by ensuring their height remains logarithmic with respect to the number of nodes. Among them, AVL trees are height-balanced binary search trees that use rotations to maintain balance during insertions and deletions. B-Trees are multi-way balanced search trees designed for efficient disk storage and are commonly used in databases and file systems. B+ Trees, an enhancement of B-Trees, store data only in leaf nodes and maintain a linked list for fast sequential access, making them ideal for range queries. Each of these trees offers specific advantages depending on the use case, and their properties and operations such as searching, insertion, and deletion are optimized to support fast and reliable data management in large-scale applications.

Recap

- ◆ A balanced binary tree ensures that the height difference between the left and right subtrees is minimal, typically $O(\log n)$.
- ◆ AVL tree is the first self-balancing binary search tree, named after Adelson-Velsky and Landis.
- ◆ In an AVL tree, the balance factor of each node must be -1, 0, or +1.
- ◆ AVL trees maintain balance using rotations after insertion and deletion operations.
- ◆ The four types of AVL rotations are Left Rotation, Right Rotation, Left-Right Rotation, and Right-Left Rotation.
- ◆ AVL trees offer efficient search, insertion, and deletion operations in $O(\log n)$ time.
- ◆ B-Trees are balanced multi-way search trees where nodes can have more than two children.
- ◆ B-Trees are widely used in databases and file systems due to their ability to minimize disk reads and writes.
- ◆ A B-Tree of order m can have a maximum of $m - 1$ keys and m children.
- ◆ All leaf nodes in a B-Tree appear at the same level, ensuring height balance.
- ◆ B+ Trees are an advanced form of B-Trees where data is stored only in leaf nodes.
- ◆ In B+ Trees, internal nodes act as index nodes and do not store actual data values.
- ◆ Leaf nodes in a B+ Tree are linked using pointers, enabling fast range queries and sequential access.

- ◆ Search operations in AVL, B, and B+ Trees follow their respective structure and offer logarithmic time complexity.
- ◆ Insertion in AVL Trees may trigger rotations to maintain balance.
- ◆ Insertion and deletion in B-Trees and B+ Trees may cause node splits, merges, or key redistribution.
- ◆ AVL Trees are ideal for in-memory structures with frequent insertions and lookups.
- ◆ B-Trees are optimized for disk-based storage systems, minimizing I/O operations.
- ◆ B+ Trees are commonly used in database indexing, especially when range queries and ordered scans are frequent.
- ◆ Understanding the properties, operations, and use cases of these balanced trees helps in choosing the right data structure for real-world applications.

Objective Type Questions

1. What is the maximum balance factor allowed in an AVL tree?
2. Which data structure is a self-balancing binary search tree?
3. What is the time complexity for searching in an AVL tree?
4. In AVL trees, what is used to maintain balance?
5. A rotation in AVL tree is performed to maintain:
6. In a B-Tree of order m , the maximum number of children a node can have is:
7. All leaf nodes in a B-Tree appear at:
8. Which operation may cause a node split in a B-Tree?
9. In B+ Trees, where is actual data stored?
10. In B+ Trees, internal nodes store:
11. Which tree supports linked leaf nodes for sequential access?
12. Which of the following is most suitable for database indexing?
13. Which type of rotation is required for left-right imbalance in AVL tree?
14. What is the balance factor of a node with equal height subtrees?
15. Which tree is best for minimizing disk I/O?
16. AVL trees are best suited for:

17. In B-Trees, keys in nodes are kept in:
18. What is the time complexity of insertion in a B+ Tree?
19. What does the B in B-Tree stand for?
20. Which of the following operations is not typically affected by AVL rotations?
21. B+ Trees improve search performance by:
22. Which tree stores data only at the leaf nodes?
23. In a B-Tree, each internal node with n keys has how many children?
24. Which of the following trees allows duplicate keys at the leaf level?
25. AVL Tree is best for which type of application?

Answers to Objective Type Questions

1. +1
2. AVL Tree
3. $O(\log n)$
4. Rotations
5. Balance of the tree
6. m
7. Same level
8. Insertion
9. Leaf nodes
10. Keys only
11. B+ Tree
12. B+ Tree
13. Left rotation followed by right rotation
14. 0
15. B-Tree
16. In-memory sorting and searching
17. Sorted order

18. $O(\log n)$
19. Balanced
20. Traversal
21. Linking leaves and separating index from data
22. B+ Tree
23. $n + 1$
24. B+ Tree
25. Applications needing fast search with frequent updates

Assignments

1. Explain in detail the structure, properties, and balancing mechanism of AVL trees. Illustrate your answer with examples showing insertion operations and the necessary rotations (LL, RR, LR, RL).
2. Describe the B-Tree data structure with its key properties and operations. Explain how insertion and deletion are handled in B-Trees and discuss how balance is maintained. Include a suitable diagram.
3. Discuss the structure and working of B+ Trees. How do B+ Trees differ from B-Trees in terms of data storage, internal nodes, and efficiency of range queries? Support your answer with diagrams.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson Education.
3. Sahni, S. (2011). Data Structures, Algorithms, and Applications in C++ (2nd ed.). Universities Press.
4. Lafore, R. (2002). Data Structures and Algorithms in Java (2nd ed.). Sams Publishing.
5. Horowitz, E., Sahni, S., & Mehta, D. (2007). Fundamentals of Data Structures in C++ (2nd ed.). Universities Press.

Suggested Reading

1. GeeksforGeeks – Data Structures <https://www.geeksforgeeks.org/data-structures/>
2. Programiz – Tree Data Structures <https://www.programiz.com/dsa/tree>
3. TutorialsPoint – Data Structures https://www.tutorialspoint.com/data_structures_algorithms/index.html
4. Open Data Structures – Online Book <https://opendatastructures.org/>

SGOU

Unit 4

Graphs

Learning Outcomes

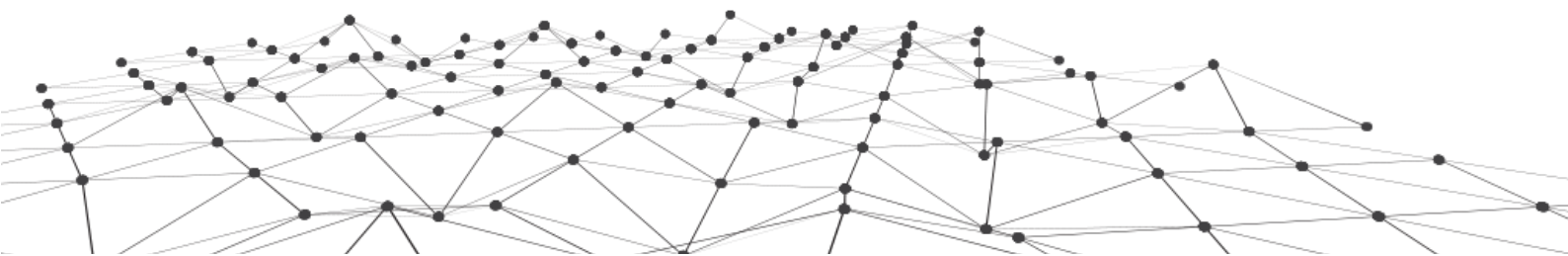
After the successful completion of the unit, the learner will be able to:

- ◆ study different graph types and basic terminologies of graphs.
- ◆ know the concept of adjacency matrix and adjacency list representation of graphs.
- ◆ learn the implementation of graphs.
- ◆ get an idea of the need for graph traversal.
- ◆ obtain the concept of depth first search and breadth first search.
- ◆ recognize an idea of applications of graph data structures.

Prerequisites

In the previous block, we studied about tree which is also a non linear data structure. There is a hierarchical relationship between parent and children in tree data structure. That is one parent and many children. But in a graph the relationship is less restricted. That is the relationship is from many parents to many children. Every tree is a graph but not conversely. Graphs are data structures which have wide ranging applications in real life like Airlines, analysis of electrical circuits, finding shortest routes, flow charts of a program etc. Many real world problems can be modeled using a graph. Graphs can be used to represent any collection of objects having some kind of pairwise relationship. Consider a city route map in your area. It may include several roads and traffic junctions.

Let's consider an example route map as shown in Figure 4.4.1 (a). We can see several roads and junctions here. Now select a particular area from the map. In Figure 4.4.1 (b) we can see the selected area shown in a rectangle. We can represent junctions as nodes and the roads as edges. Here six junctions are there in the selected area. All the six junctions are represented using nodes. The roads connecting these junctions are represented by edges. Thus a small area of map is represented by a graph which is shown in Figure 4.4.1 (c). So a graph contains a set of nodes or vertices and edges or links. An edge connects two vertices.



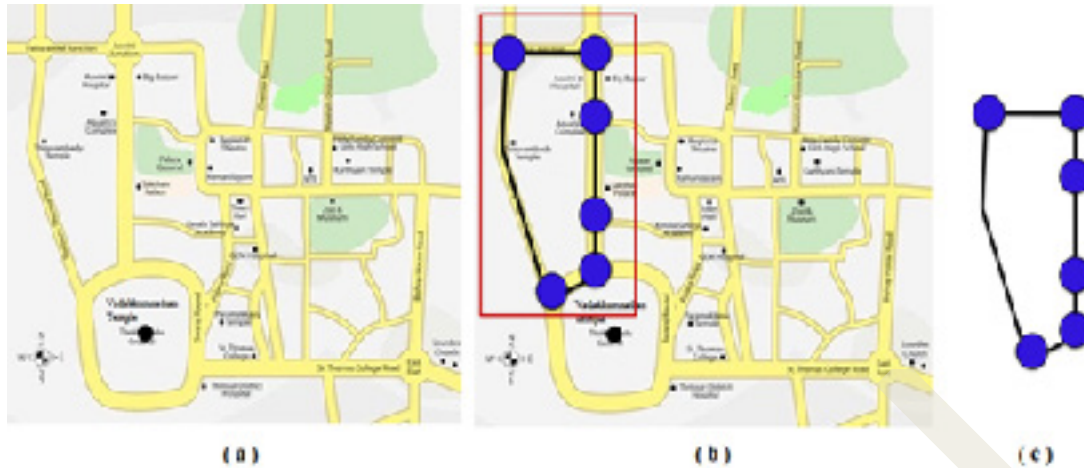


Fig 4.4.1 Route Map

Key words

Undirected and Directed Graphs, Weighted Graph, Vertices, Edges, Regular and Planar Graphs, Cyclic and Acyclic Graphs, Connected and Disconnected Graphs, Biconnected Graph, Graph Representations, Graph Traversal, Applications of Graph.

Discussion

4.4.1 Definition of Graph

A graph G consists of a non empty finite set of vertices $V(G)$ and a finite set of edges $E(G)$; where $V(G) = \{v_0, v_1, \dots, v_n\}$ or set of vertices or nodes and $E(G) = \{e_1, e_2, \dots, e_n\}$ or set of edges.

The set of edges contains a pair of vertices. If $e = (v_i, v_j)$ is an edge with vertices v_i and v_j , and $v_i, v_j \in V(G)$, then v_i and v_j are said to lie on edge, e and e is said to be incident with v_i and v_j .

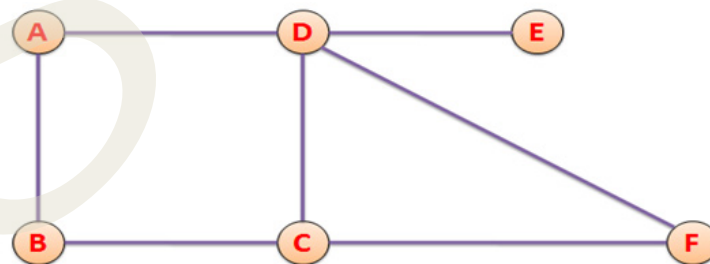


Fig 4.4.2 Example of simple Graph.

Consider a simple graph G with vertices A, B, C, D, E, and F as shown in Figure 4.4.2. We can see that there are 6 vertices or nodes and 7 edges. That is,

$$V(G) = \{A, B, C, D, E, F\}$$

$$E(G) = \{(A, B), (A, D), (B, C), (D, C), (C, F), (D, E), (D, F)\}$$

In the edge set $E(G)$,

- ◆ (A, B) represents the edge between the nodes A, B.
- ◆ (A, D) represents the edge between the nodes A, D.
- ◆ (B, C) represents the edge between the nodes B, C.
- ◆ (D, C) represents the edge between the nodes D, C.
- ◆ (C, F) represents the edge between the nodes C, F.
- ◆ (D, E) represents the edge between the nodes D, E.
- ◆ (D, F) represents the edge between the nodes D, F.

Problem 1: Draw a simple graph with 5 vertices and 4 edges whose vertices are P, Q, R, S, and T. The edges of the graph are PQ, QR, PS, and ST.

Solution: First, represent 5 vertices P, Q, R, S, and T with 5 different nodes (Figure 4.4.3). For representing a node, we use small circles. Each circle is labeled with a name: P, Q, R, S, and T are the labels. In the next step, add each edge by connecting the corresponding vertices. Take the first edge, PQ. The vertices are P and Q, so connect node P and node Q with a line. Then add the next edge, QR, followed by PS, and finally ST. This completes the graph. (Fig 4.4.3)



Fig 4.4.3 The resultant graph with 5 vertices and 4 edges.

4.4.2 Types of Graphs

Graphs are essential structures in computer science and mathematics used to model relationships between pairs of objects. A graph is made up of vertices (also called nodes) connected by edges, and it provides a visual and logical way to represent networks such as social media connections, transportation systems, or computer networks. There are various types of graphs, each serving different purposes depending on the nature of the data and the relationships involved. Understanding the different types of graphs is crucial for designing efficient algorithms and solving complex problems in diverse fields. A graph can be broadly classified into two types: Undirected graph and Directed graph.

a. Undirected Graph

If the pair of vertices is unordered then graph G is called an undirected graph. Which means if there is an edge between v_1 and v_2 then it can be represented as (v_1, v_2) or (v_2, v_1) . Fig. 4.4.4 shows an example for an undirected graph. It consists of 5 vertices and 5 edges. That is,

$$V(G) = \{ A, B, C, D, E \}$$

$$E(G) = \{ (A, B), (A, C), (A, D), (D, C), (C, E) \}$$

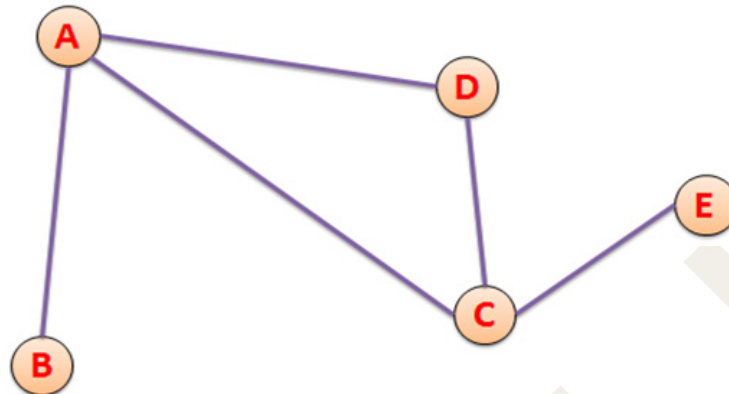


Fig 4.4.4 Example of Undirected Graph

Here the edge (A, B) can be represented as (B, A) also. The edge (A, C) can be represented as (C, A). Similarly the other 3 edges can be represented as (D, A), (C, D), and (E, C). The pairs of vertices are unordered. So we can call it an undirected graph.

b. Directed Graph:

If the pair of vertices are ordered, then the graph G is called a directed graph or a digraph. Which means that a graph has an ordered pair of vertices (v_1, v_2) , where v_1 is the tail and v_2 is the head of the edge; we can call it a digraph. The line segments or arcs (edges) of the directed graphs have arrowheads which indicates the direction. Figure 4.4.5 shows an example of a directed graph. It has 5 vertices and 6 edges.

That is,

$$V(G) = \{ A, B, C, D, E \}$$

$$E(G) = \{ (A, D), (D, E), (E, C), (C, D), (B, C), (B, A) \}$$

Here, we can see that the edge (A, D) is represented with an arrow headed line whose tail is A and head of the edge is D. We can't represent the same edge as (D, A) because the vertex D is not the tail and the vertex A is not the head of the edge. Likewise all other edges are represented in this manner.

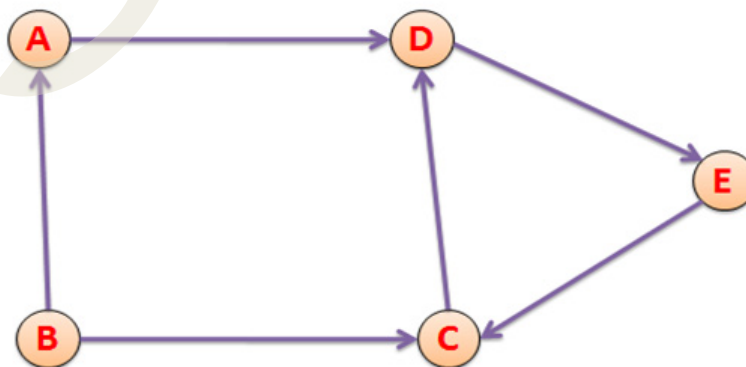


Fig 4.4.5 Example of directed Graph

4.4.3 Terminologies used in Graphs

Graphs are widely used to represent complex relationships between objects, and understanding their structure requires familiarity with specific terminologies. These terms help describe the components and properties of a graph, enabling clear communication and effective problem-solving. Key graph terminologies include vertices (or nodes), edges (or arcs), degree, adjacency, path, cycle, and connectivity, among others. Each of these terms plays a vital role in defining the behavior and characteristics of different types of graphs. Gaining a strong grasp of graph terminology is fundamental for studying graph theory and applying it in areas such as computer networks, data structures, social networks, and transportation systems.

4.4.3.1 Weighted Graph

Consider a city route map which is represented by a graph. The junctions are represented using nodes and the roads connecting them are represented using edges. Take an edge connecting two nodes (vertices). We can label this edge with a value that corresponds to the distance between those nodes. We can say it as the weight of that edge. Similarly the speed limit in a road (edge in the graph) can be represented as weight.

If all the edges in a graph are labeled with some numbers or weights, then the graph is called a weighted graph. Figure 4.4.6 shows an example for a weighted graph. Here P, Q, R, and S are the vertices of the graph. The graph consists of 4 edges and all are labeled with some weights. Edge PQ is labeled with a weight 4. Edge PR is labeled with a weight 2. Edge RS is labeled with a weight 5 and edge QS is labeled with a weight 3 in figure 4.4.6.

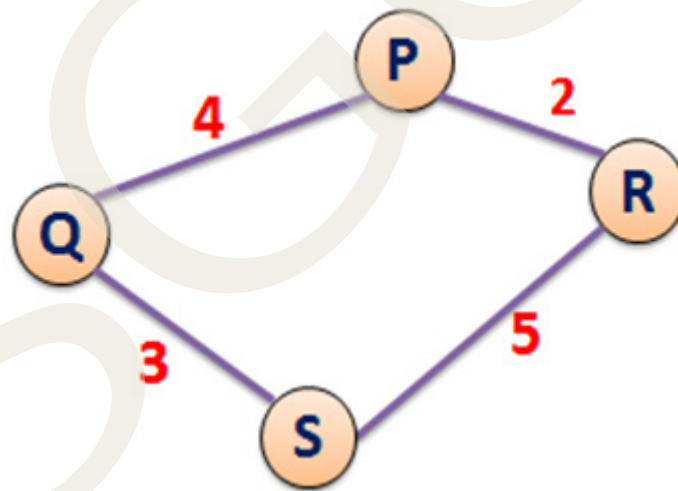


Fig 4.4.6 An example for Weighted Graph

4.4.3.2 Self Loop

For a graph G , if there is an edge whose starting and ending vertices are the same, that is $(v1, v1)$ then it is called a self loop. Figure 4.4.7 shows an example of a graph with a self loop. Here $v1, v2, v3$, and $v4$ are the vertices of the graph. The edges are $(v1, v2)$, $(v2, v3)$, $(v3, v4)$, $(v4, v1)$ and $(v1, v1)$ and the edge $(v1, v1)$ is a self loop.

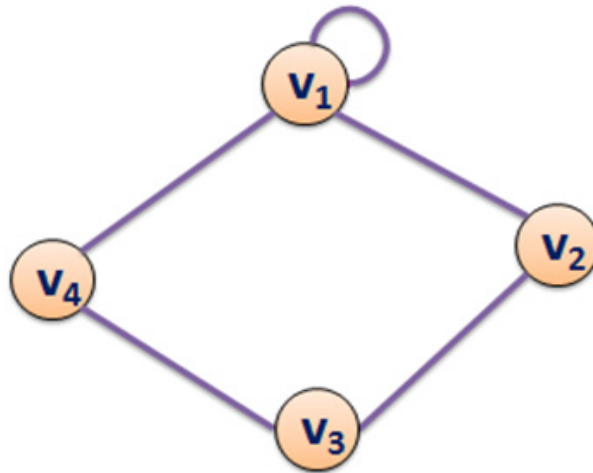


Fig 4.4.7 An example for Self Loop

4.4.3.3 Parallel Edges

For a graph G , if there are more than one edges between the same pair of vertices then they are known as parallel edges. Figure 4.4.8 shows a graph with parallel edges. We can see there are two edges between the same pair of vertices v_1 and v_2 .

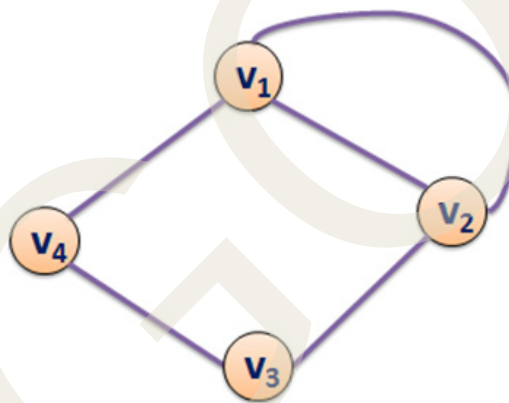


Fig 4.4.8 An example for Parallel Edges

4.4.3.4 Adjacent Vertices

In a graph G , a vertex u is adjacent to another vertex v if there is an edge from u to v . In an undirected graph if (v_1, v_2) is an edge then v_1 is adjacent to v_2 and v_2 is adjacent to v_1 . In a directed graph if (v_1, v_2) is an edge then v_1 is adjacent to v_2 and v_2 is adjacent from v_1 . For example, Figure 4.4.9 (i) shows an undirected graph, vertex u is adjacent to vertex v and vertex v is adjacent to vertex u . Figure 4.4.9 (ii) shows a directed graph, vertex u is adjacent to vertex v and vertex v is adjacent from vertex u .



Fig 4.4.9 An example for adjacent vertices and Incidence

4.4.3.5 Incidence

In an undirected graph the edge (u, v) is incident on vertices u and v . In a directed graph the edge (u, v) is incident from node u and is incident to node v .

For example, Figure 4.4.9(i) shows an undirected graph; the edge (u, v) is incident on vertices u and v . Figure 4.4.9(ii) shows a directed graph; edge (u, v) is incident from node u and is incident to node v .

4.4.3.6 Degree of Vertex

The degree of a vertex is the number of edges incident to that vertex. The degree of a node in an undirected graph is the number of edges connected to that node.

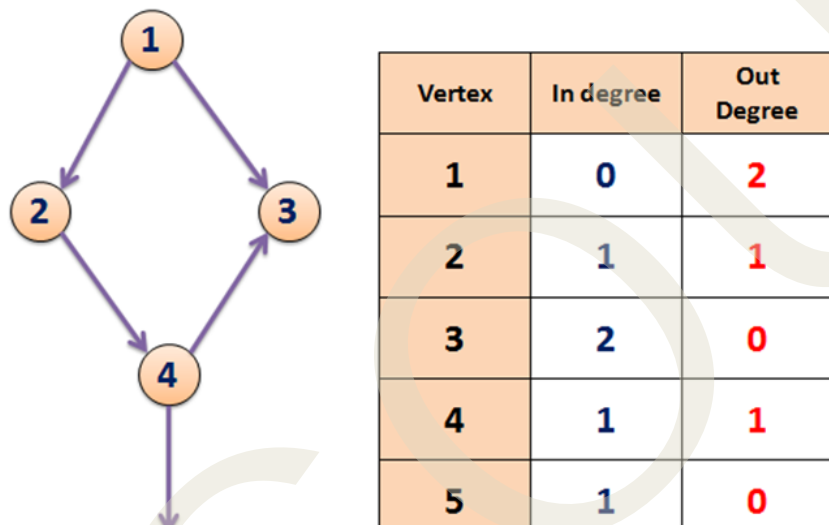


Fig 4.4.10 Degree of Vertex in an Undirected Graph

Take an example graph shown in Figure 4.4.10, to find the degree of a vertex. Here the undirected graph contains five vertices. We can find the degree of each vertex in Figure 4.4.10.

The degree of vertex 1 is 2 because 2 edges are incident to vertex 1.

The degree of vertex 2 is 3 because 3 edges are incident to vertex 2.

The degree of vertex 3 is 2 because 2 edges are incident to vertex 2.

The degree of vertex 4 is 2 because 2 edges are incident to vertex 4.

The degree of vertex 5 is 1 because only 1 edge is incident to the vertex 5.

In a directed graph there are two types of degrees for every node; in degree and out degree.

In degree: The in degree of a vertex is the number of edges coming to that vertex or edges incident to it.

Out degree: The out degree of a vertex is the number of edges going outside from that node or the edges incident from it.

Take an example digraph shown in Fig. 4.4.11, to find the degree of a vertex. Here the directed graph contains five vertices. We can find the in degree and out degree of each vertex in Figure 4.4.11.

The **in degree** of vertex 1 is **0** because no edges are incident to vertex 1.

The **in degree** of vertex 2 is **1** because only 1 edge is incident to the vertex 2.

The **in degree** of vertex 3 is **2** because 2 edges are incident to vertex 3.

The **in degree** of vertex 4 is **1** because only 1 edge is incident to the vertex 4.

The **in degree** of vertex 5 is **1** because only 1 edge is incident to the vertex 5.

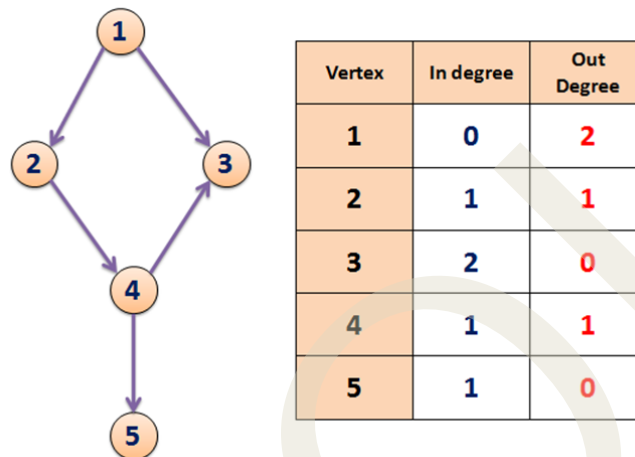


Fig 4.4.11 Degree of Vertex in a Directed Graph

The **out degree** of vertex 1 is **2** because 2 edges are incident from the vertex 1.

The **out degree** of vertex 2 is **1** because only 1 edge is going outside from the vertex 2.

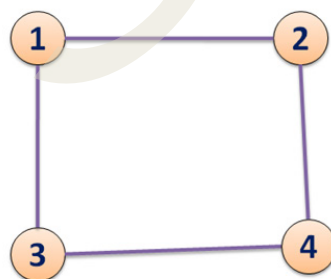
The **out degree** of vertex 3 is **0** because no edges are going outside from the vertex 3.

The **out degree** of vertex 4 is **1** because only 1 edge is incident from the vertex 4.

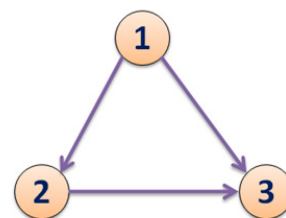
The **out degree** of vertex 5 is **0** because no edges are going outside from the vertex 5.

4.4.3.7 Simple graph

A graph or digraph which does not have only self loop or parallel edges is called a simple graph. Figure 4.4.12 shows examples for simple graph. (i) shows a simple undirected graph. There are no self loops and parallel edges. (ii) shows a simple digraph with no self loops and parallel edges.



(i) A simple undirected graph



(ii) A simple directed graph

Fig 4.4.12 Examples for Simple Graph

4.4.3.8 Multi Graph

A graph which has either a self loop or parallel edges or both is called a multi graph.

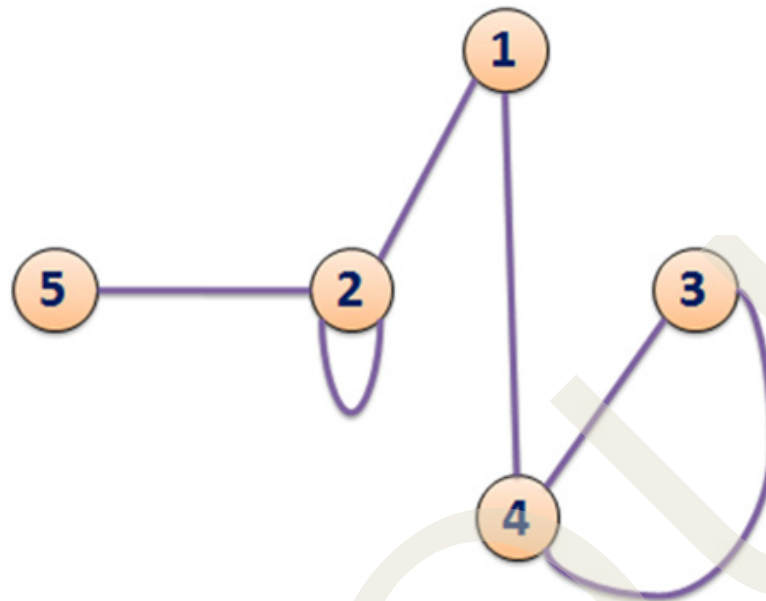


Fig 4.4.13 Example for Multi Graph

Figure 4.4.13 shows an example for a multi graph. The graph consists of 5 vertices and 6 edges. There is a self loop in node 2. The vertices 4 and 3 are connected with 2 parallel edges. So it is a multi graph.

4.4.3.9 Maximum Edges in Graph

In a simple undirected graph there can be $n(n-1)/2$ maximum edges and in a simple directed graph there can be $n(n-1)$ maximum edges. Where n is the total number of vertices in the graph.

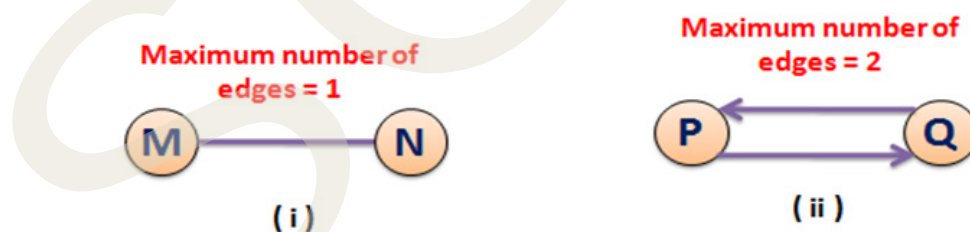


Fig 4.4.14 Examples for Maximum edges

For example (Figure 4.4.14 (i)), if the number of nodes of a simple undirected graph is 2, then the maximum number of edges will be 1. That is $(2 \times (2-1))/2 = 1$.

For example (Figure 4.4.14(ii)), if the number of nodes of a simple directed graph is 2, then the maximum number of edges will be $(2 \times 1) = 2$.

4.4.3.10 Complete Graph

A graph is said to be complete if each vertex is adjacent to every other vertex in the graph.

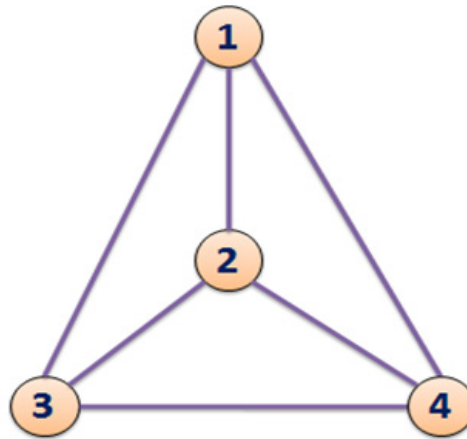


Fig 4.4.15 Example for Complete Graph

Figure 4.4.15 shows an example for a complete graph. In figure we can see four vertices; say 1, 2, 3, and 4. Vertex 1 is adjacent to vertices 2, 3, and 4. Vertex 2 is adjacent to vertices 1, 3, and 4. Vertex 3 is adjacent to vertices 1, 2, and 4. Vertex 4 is adjacent to vertices 1, 2, and 3. So each vertex is adjacent to every other vertex in the graph. So we can call it a complete graph. The total number of edges in an undirected complete graph will be $\frac{n(n-1)}{2}$; where n is the total number of vertices or nodes

In Figure 4.4.15, $n = 4$. So the total number of edges will be $\frac{(4*(4-1))}{2}$; ie. $\frac{12}{2} = 6$ edges.

4.4.3.11 Regular Graph

A graph is regular if every node is adjacent to the same number of nodes. That is, each node in the graph has the same degree. Figure 4.4.16 shows an example of a regular graph. It consists of 10 nodes. Each node is adjacent to the same number of nodes. That is, each node has the same degree. Take node 1. It is adjacent to nodes 2, 3 and 6. Therefore the degree of node 1 is 3. Node 2 is adjacent to nodes 1, 4 and 7. The degree of node 2 is 3.

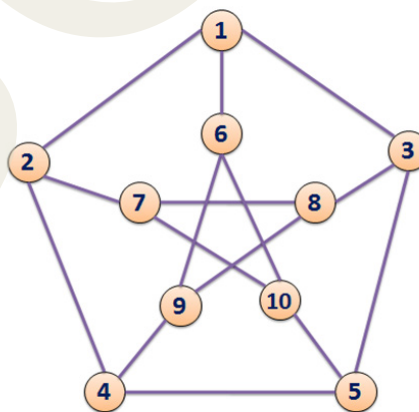


Fig 4.4.16 Example for Regular Graph

Likewise we can see that all other nodes in the graph have the same degree. That is the degree of nodes 3, 4, 5, 6, 7, 8, 9 and 10 is equal to 3. So it is a regular graph.

4.4.3.12 Planar Graph

A graph is planar if it can be drawn in a plane without any two edges intersecting.

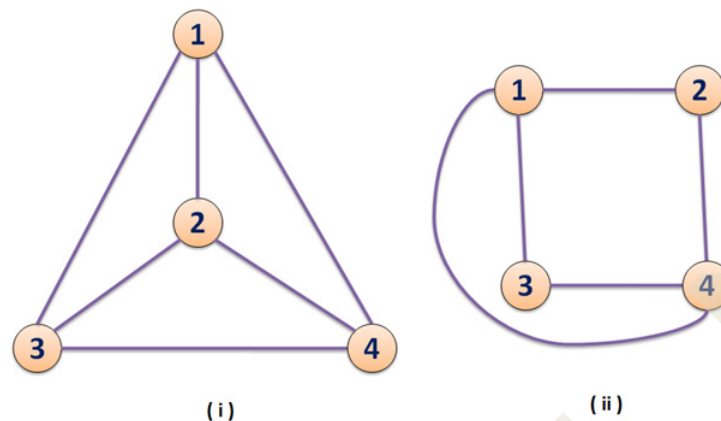


Fig 4.4.17 Examples for Planar Graphs

Figure 4.4.17 shows examples for planar graphs. In both the graphs no two edges are intersecting.

4.4.3.13 Walk & Path

In a graph G , a **walk** is a finite sequence of edges of the form $v_0v_1, v_1v_2, v_2v_3, \dots, v_{n-1}v_n$. And we can also denote this by $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \dots \rightarrow v_n$ in which any 2 consecutive edges are adjacent or identical. Such a walk determines a sequence of vertices $v_0, v_1, v_2, \dots, v_n$. We can call v_0 as the initial vertex and v_n as the final vertex of the walk. That is a walk from v_0 to v_n .

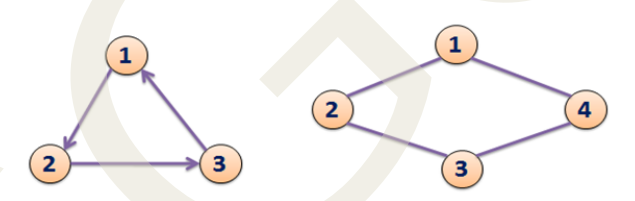


Fig 4.4.18 Example for walk and path

The number of edges in a walk called its length. For example in Figure 4.4.18, $P \rightarrow Q \rightarrow R \rightarrow S \rightarrow T \rightarrow T \rightarrow S \rightarrow Q$ is a walk of length 7 from P to Q.

If all the edges in a walk are distinct then it is called a trail. If all the vertices in a trail are distinct then it is called a path. That is the vertices in a trail, $v_0, v_1, v_2, \dots, v_n$ are distinct, then the trail is a path. If $v_0 = v_n$ then the path or trail is called a closed path or a closed trail.

Consider Figure 4.4.18 we can see that

$P \rightarrow Q \rightarrow R \rightarrow S \rightarrow T \rightarrow T \rightarrow R$ is a trail.

$P \rightarrow Q \rightarrow R \rightarrow S \rightarrow T$ is a path.

$P \rightarrow Q \rightarrow R \rightarrow S \rightarrow T \rightarrow R \rightarrow P$ is a closed trail.

$P \rightarrow Q \rightarrow S \rightarrow T \rightarrow R \rightarrow P$ is a closed path.

4.4.3.14 Cycle

If there is a path containing one or more edges which starts from a vertex and terminates into the same vertex then the path is called a cycle. A closed path containing at least one edge is a cycle. Any loop or pair of multiple edges is a cycle.

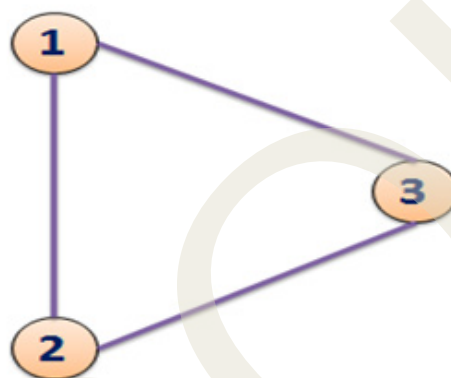


Fig 4.4.19 Examples for a Cycle

Figure 4.4.19 shows an example for a cycle. Take the vertex 1, we can see a path $1 - 2 - 3 - 1$. That is a path starting from 1 and terminating at 1. So it is a cycle.

4.4.3.15 Cyclic Graph

A graph that has cycles is called a cyclic graph. In Figure 4.4.20 we can see both directed and undirected graphs with a cycle.

· The 1st digraph is cyclic because it contains a cycle $1 - 2 - 3 - 1$.

The 2nd undirected graph is also cyclic because it contains a cycle $1 - 2 - 3 - 4 - 1$.

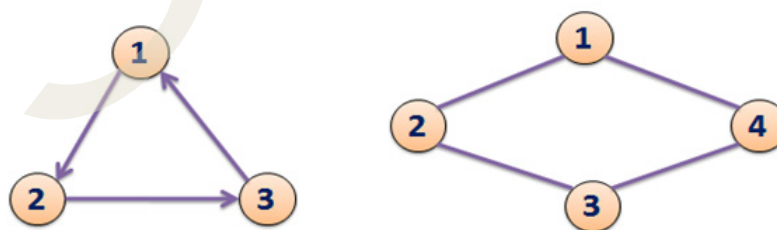


Fig 4.4.20 Examples for Cyclic Graph

4.4.3.16 Acyclic Graph

If a graph does not have any cycle then it is called an acyclic graph. In Figure 4.4.21 we

can see both directed and undirected graphs with no cycles. The first digraph is acyclic because it contains no cycle. It contains the following paths;

1 - 2 - 3, 1 - 3, 2 - 3

All the three paths are not closed. So there is no cycle. That is the digraph is acyclic.



Fig 4.4.21 Examples for Acyclic Graph

The second undirected graph shown in Figure 4.4.21, is also acyclic because it contains no cycle. It contains the following paths;

1 - 2 - 3, 3 - 2 - 1, 2 - 1, 1 - 2, 2 - 3, 3 - 2

All the six paths are not a cycle. So the graph is acyclic.

4.4.3.17 Connected Graph

In a graph G , two vertices v_1 and v_2 are said to be connected if there is a path in G from v_1 to v_2 or v_2 to v_1 . A graph is said to be connected if there is a path from any node of graph to any other node. That is, for every pair of distinct vertices in G , there exists a path. Figure 4.4.22 (i) shows a connected graph. Here we can see that each pair of nodes is connected. There is a path from node 1 to nodes 2, 3 and 4. Likewise all other nodes have a path to every other node. Figure 4.4.22 (ii) shows a disconnected graph. The graph is disconnected because there is no path to node 4 from the nodes 1, 2 and 3.

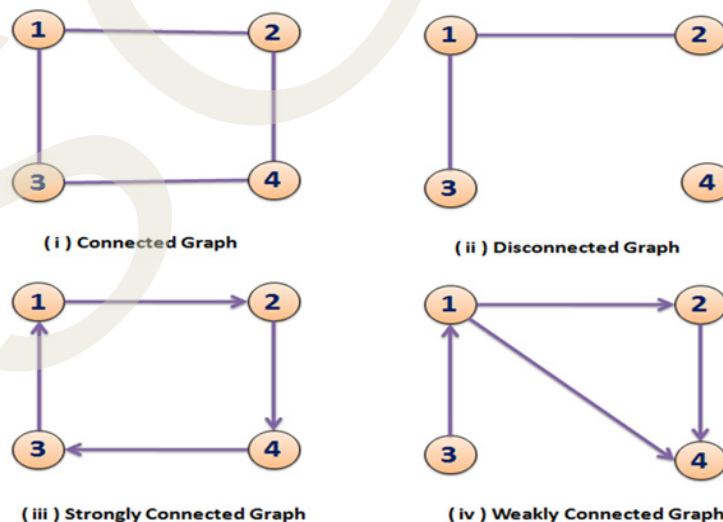


Fig 4.4.22 Examples for Connected and Disconnected Graphs

A directed graph G is said to be strongly connected, if every pair of distinct vertices in G , there is a path. Figure 4.4.22 (iii) shows a strongly connected digraph. For example

take the case of vertex 1; there is a path from 1 to 2, 1 to 3, and 1 to 4. In the case of vertex 2; there is a path from 2 to 3, 2 to 4, and 2 to 1. Likewise in the case of vertex 3 and vertex 4; there is a path to all other vertices.

A digraph is called weakly connected, if for any pair of vertices u and v , there is a path from u to v or a path from v to u . In a digraph, if we replace the directed edge with undirected edges and the resulting graph is connected then that digraph is weakly connected. Figure 4.4.22 (iv) shows a weakly connected graph. In the case of vertex 1; there is a path from 1 to vertices 2 and 4, and a path from vertex 3 to 1. If we replace all the directed edges with undirected edges here, then the resultant graph is a connected one. That is there is a path from vertex 1 to vertices 2, 3, and 4. There is a path from vertex 2 to vertices 1, 3, and 4. There is a path from vertex 3 to vertices 1, 2, and 4. There is a path from vertex 4 to vertices 1, 2, and 3. So the graph is weakly connected.

4.4.3.18 Articulation Point

If on removing a node from the graph, the graph becomes disconnected then that node is called the articulation point.

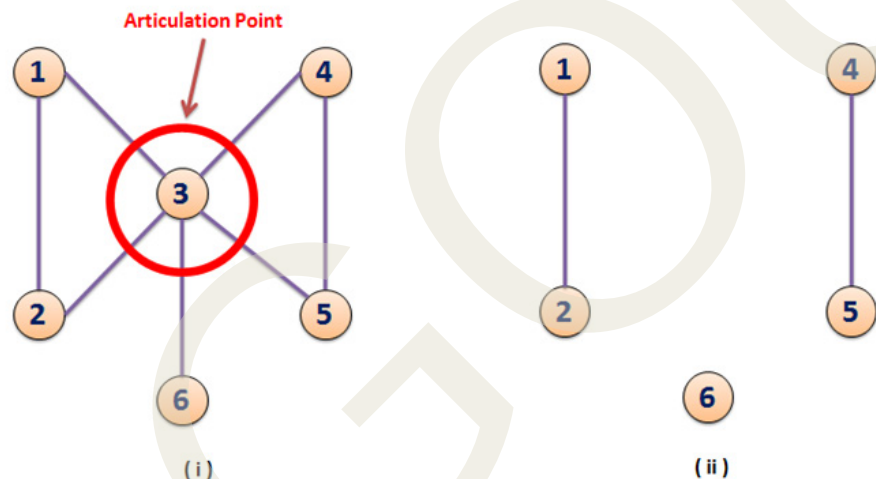


Fig 4.4.23 Example for Articulation point in a graph

In Figure 4.4.23 (i), node 3 is an articulation point because on removing node 3, the graph becomes disconnected which is shown in Figure 4.4.23 (ii). On removing node 3 from the graph, all the 5 edges connected to node 3 are removed. That is, edges 3-1, 3-2, 3-6, 3-5 and 3-4 are removed. Then the resultant graph becomes disconnected.

4.4.3.19 Bridge

If on removing an edge from the graph, the graph becomes disconnected then that edge is called the bridge.

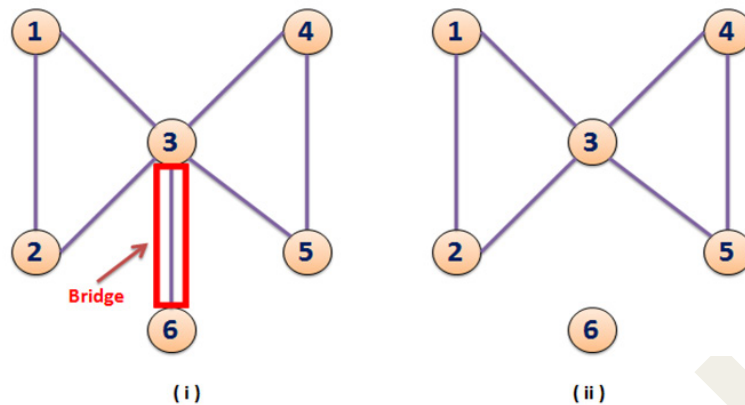


Fig 4.4.24 Example for Bridge in a graph

In Figure 4.4.24 (i) edge 3-6 is a bridge because if this edge is removed then the graph becomes disconnected which is shown in Figure 4.4.24 (ii).

4.4.3.20 Biconnected Graph

A graph with no articulation points is called a biconnected graph.

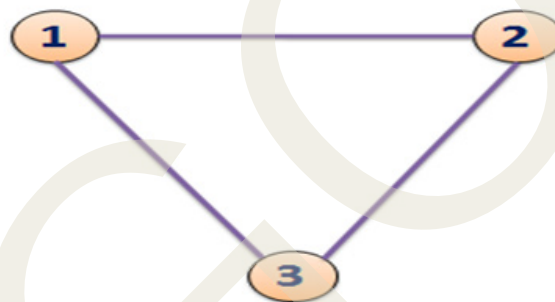


Fig 4.4.25 Example for Biconnected Graph

In Figure 4.4.25, there is no articulation point. There are three vertices and three edges in the given graph. Try to remove vertex 1 from the graph which does not make the graph disconnected.

By removing vertex 2 does not make the graph disconnected. By removing vertex 3 does not make the graph disconnected. So it is a biconnected graph.

4.4.4 Graph Representation

Graph representation refers to the method used to store and organize graph data in a way that allows efficient access and manipulation. Since graphs consist of vertices and edges, their representation in memory plays a crucial role in the performance of graph-related algorithms. The two most common techniques for representing graphs are the **adjacency matrix** and the **adjacency list**, each with its own advantages depending on the graph's density and operations performed. Choosing the right representation is essential for solving problems related to shortest paths, connectivity, traversal, and more. A clear understanding of graph representation lays the foundation for implementing and analyzing graph algorithms effectively.

4.4.4.1 Adjacency Matrix Representation of Graph

Adjacency matrix is the matrix that keeps the information of adjacent nodes. That is, the matrix keeps the information that whether the vertex is adjacent to any other vertex or not. It is a sequential representation method. The general representation of the adjacency matrix is shown in Figure 4.4.26.

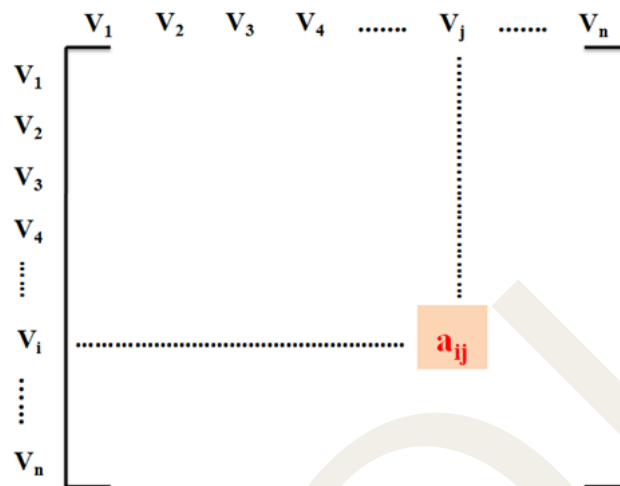


Fig 4.4.26 Adjacency Matrix Representation

Here $V_1, V_2, V_3, \dots, V_n$ are the vertices of the given graph and a_{ij} represents the edge from V_i to V_j . The value of a_{ij} is either 1 or 0 according to the rule which is given below.

$a_{ij} = 1$; if there is an edge from V_i to V_j .
0; otherwise

This adjacency matrix is also called a Bit matrix or Boolean matrix because the entries are either 0 or 1.

Example 1: Consider the following digraph in Figure 4.4.27. From this we can write the corresponding adjacency matrix.

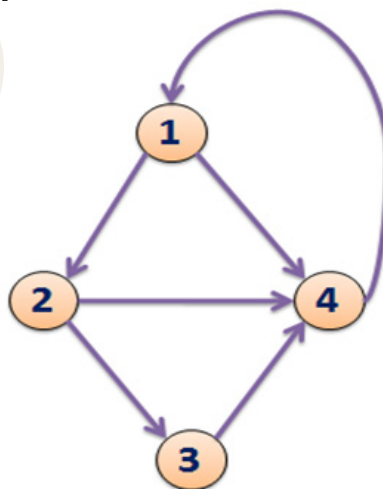


Fig 4.4.27 Example 1

There are 6 directed edges in this graph. First take the vertex 1. There is an edge from vertex 1 to 2. So we can represent $a_{12} = 1$. There is an edge from vertex 1 to 4. That is $a_{14} = 1$. In the case of vertex 2, there is an edge from 2 to 3 and 2 to 4. That is $a_{23} = 1$, and $a_{24} = 1$. In the case of vertex 3, there is an edge from 3 to 4. That is $a_{34} = 1$. In the case of vertex 4, there is an edge from 4 to 1. That is $a_{41} = 1$. All the other entries of the adjacency matrix will be 0. Then we can represent the adjacency matrix as;

1 2 3 4 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0

Example 2: Consider the following undirected graph in Figure 4.4.28. From this we can write the corresponding adjacency matrix. There are 6 edges in this graph

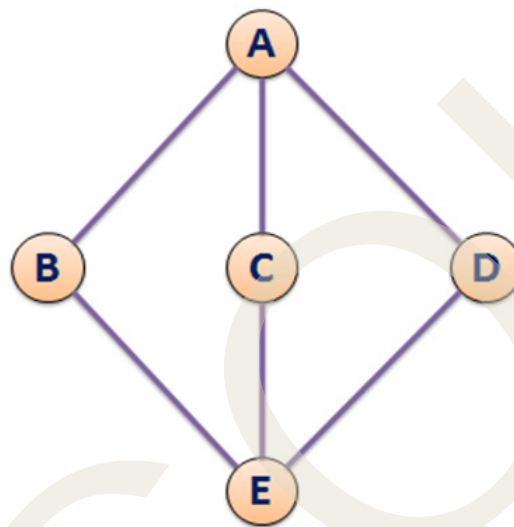


Fig 4.4.28 Example 2

First take the vertex A. There is an edge from vertex A to B. So we can represent $a_{AB} = 1$. There is an edge from vertex A to C and A to D. That is $a_{AC} = 1$, and $a_{AD} = 1$. In the case of vertex B, there is an edge from B to E and B to A. That is $a_{BE} = 1$, and $a_{BA} = 1$. In the case of vertex C, there is an edge from C to A and C to E. That is $a_{CA} = 1$ and $a_{CE} = 1$. In the case of vertex D, there is an edge from D to A and D to E. That is $a_{DA} = 1$ and $a_{DE} = 1$. In the case of vertex E, there is an edge from E to B, E to C, and E to D. So $a_{EB} = 1$, $a_{EC} = 1$ and $a_{ED} = 1$. All the other entries of the adjacency matrix will be 0. Then we can represent the adjacency matrix as;

A B C D E 0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0

Note:

- ♦ The adjacency matrix of an undirected graph is symmetric but not conversely. If A is the adjacency matrix of a simple undirected graph, then $A = A^T$; where A^T is the transpose of A.
- ♦ From the adjacency matrix of an undirected graph, the degree of any vertex i is its row sum.
- ♦ From the adjacency matrix of a directed graph, the out degree of any vertex i is its row sum and in degree is its column sum.

- ♦ The space needed to represent a graph using its adjacency matrix is n^2 bits.
- ♦ In the case of multi graphs instead of entry 1, the entry will be the number of edges between two vertices.
- ♦ For a weighted graph the entries in the adjacency matrix are the weights of the edges between the vertices.

Example 3: Consider the following weighted digraph in Figure 4.4.29. From this we can write the corresponding adjacency matrix. There are 5 directed edges in this graph.

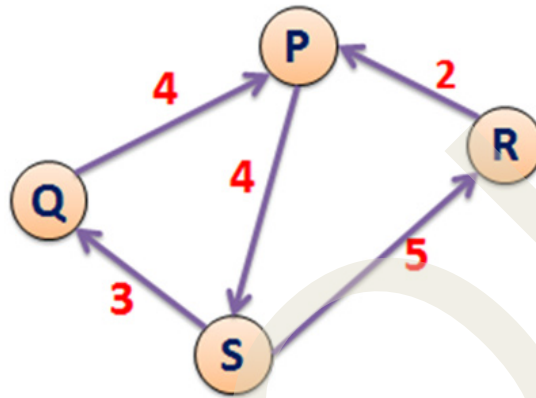


Fig 4.4.29 Example 3

First take the vertex P. There is an edge from vertex P to S. So we can represent $a_{PS} = 4$ because the weight of the edge PS is 4. In the case of vertex Q, there is an edge from Q to P. That is $a_{QP} = 4$ because the weight of the edge QS is 4. In the case of vertex R, there is an edge from R to P. That is $a_{RP} = 2$ because the weight of the edge RP is 2. In the case of vertex S, there is an edge from S to Q and S to R. That is $a_{SQ} = 3$ and $a_{SR} = 5$ because the weight of the edge SQ is 3 and the weight of the edge SR is 5. All the other entries of the adjacency matrix will be 0. Then we can represent the adjacency matrix as;

P Q R S 0 0 0 4 4 0 0 0 2 0 0 0 0 3 5 0

Example 4: Draw the undirected graph corresponding to the adjacency matrix.

$A(G) = 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0$

Solution: $A(G)$ is 4×4 matrix, hence G will have 3 vertices. We can take it as 1, 2, and 3. Draw an edge from v_i to v_j where $a_{ij} = 1$. Here;

$a_{12} = 1$ and $a_{21} = 1$. So draw an undirected edge from vertex 1 to vertex 2.

$a_{13} = 1$ and $a_{31} = 1$. So draw an undirected edge from vertex 1 to vertex 3.

$a_{23} = 1$ and $a_{32} = 1$. So draw an undirected edge from vertex 2 to vertex 3.

$a_{24} = 1$ and $a_{42} = 1$. So draw an undirected edge from vertex 2 to vertex 4.

The resultant graph is shown in Figure 4.4.30.

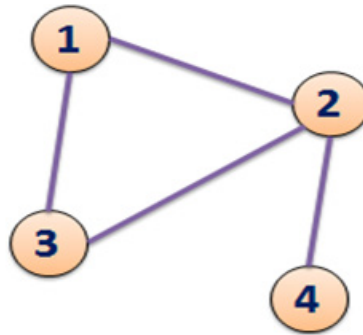


Fig 4.4.30: Example 4

4.4.5 Adjacency List Representation of Graph

Adjacency matrix representation is a sequential representation method. For a graph G with n vertices or nodes; it may be difficult to insert and delete nodes with sequential representation in memory. This is because the size of the matrix may need to be changed and the nodes may need to be reordered. So we need a better representation of G in memory. For this, we use linked representation. It is also known as adjacency list representation. In adjacency list representation, the number of lists depends on the number of vertices. For example if there are 5 nodes in the graph then in adjacency list representation consists of 5 separate lists. That is, the adjacency list represents a graph as an array of linked lists. The index of the array represents the number of vertices. Each element in the list represents the other vertices that make an edge with the vertex. The following session explains how the graphs (both the undirected and directed) are represented with an adjacency list with suitable diagrams.

4.4.5.1 Representing an Undirected Graph

Consider an example undirected graph shown in Figure 4.4.31(i). It has four vertices. So the linked representation includes 4 separate lists. We can see the adjacency list representation in Figure 4.4.31(ii). The index of the array starts with 1. So the first list represents the adjacent vertices of vertex 1. The adjacent vertices of vertex 1 are 2, 3, and 4. So the list contains four node structures. It starts from 1 then points to 2, then points to 3, then points to 4. 4 is the terminal node; which means the adjacency list of vertex 1 terminates at node 4.

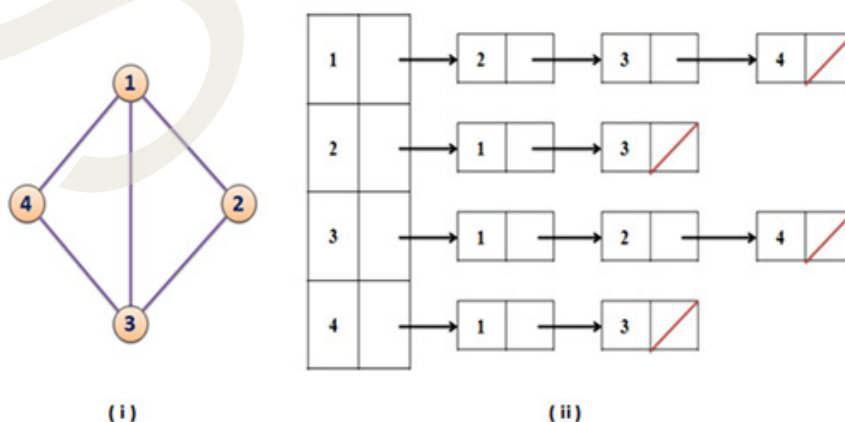


Fig 4.4.31 Adjacency List representation of an Undirected Graph

The next index is 2. That is the vertex 2. Vertex 2 has two adjacent vertices; vertex 1 and vertex 3. So index 2 points to node structure 1 which is the representation of node 1 and then points to node 3 and terminates. In the case of node 3, the adjacent nodes or vertices are node 1, node 2, and node 4. So we can represent the list as; node 3 points to 1, then points to 2, then points to 4 and terminates. In the case of node 4, the adjacent vertices are node 1 and node 3. So the list representation starts from node 4 and points to node 1 and then points to node 3 and terminates.

4.4.5.2 Representing a Directed Graph

Consider a digraph shown in Figure 4.4.32(i). The graph has 5 nodes. So the array indexes include 1, 2, 3, 4, and 5. The linked representation is shown in Figure 4.4.32(ii). We can see 5 separate lists in the adjacency list representation. In the case of node 1, the adjacent nodes are node 2 and node 3. So in the adjacency list representation, the node 1 points to 2 and then points to 3 and terminates. In the case of node 2, the adjacent node is node 3. So in the adjacency list representation, the node 2 points to 3 and terminates. In the case of node 3, the adjacent node is node 5. So in the adjacency list representation, the node 3 points to 5 and terminates.

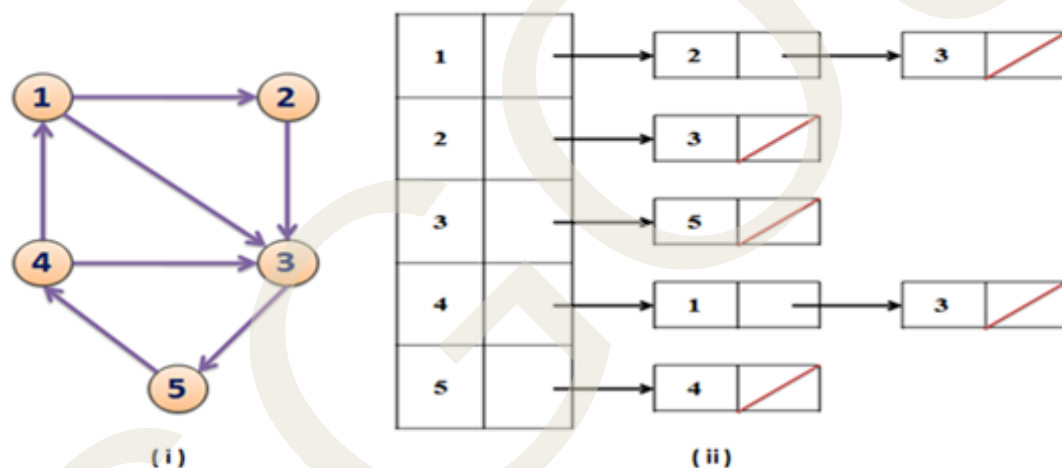


Fig 4.4.32 Adjacency List representation of a Directed Graph

In the case of node 4 the adjacent nodes are node 1 and node 3. So in the adjacency list representation, the node 4 points to 1 and then points to 3 and terminates. In the case of node 5, the adjacent node is node 4. So in the adjacency list representation, the node 5 points to 4 and terminates.

4.4.5.3 Multi List Representation of Graph

Adjacency multi list representation of a graph is based on the edges. In this graph structure there are two main parts and they are; a directory of node information, and a set of linked lists of edge information. The directory of node information is an array of header nodes. For each node of the graph there is one entry in the node directory. Each edge in the graph has a separate list representation. For representing the edge information the following structure (Figure 4.4.33) is used.

M	Vertex 1	Vertex 2	List 1	List 2
----------	-----------------	-----------------	---------------	---------------

Fig 4.4.33 Structure used for storing edge information

- ◆ M is a 1 bit field which is used to denote whether the node is examined or not.
- ◆ Vertex 1 is the start vertex of an edge (u, v). That is Vertex 1 = u.
- ◆ Vertex 2 is the end vertex of an edge (u, v). That is Vertex 1 = v.
- ◆ List 1 represents the first down “List name” where Vertex 1 is present.
- ◆ List 2 represents the first down “List name” where Vertex 2 is present.

The header nodes are represented in array format. Here we can call it as a directory of node information. The header nodes point to the corresponding edge list.

Consider a simple example to understand the multi list representation. Figure 4.4.34 shows a simple undirected graph G with four vertices 1, 2, 3, and 4. There are 6 edges in the graph.

ie. The vertex set is; $V(G) = \{ 1, 2, 3, 4 \}$

The edge set is; $E(G) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$

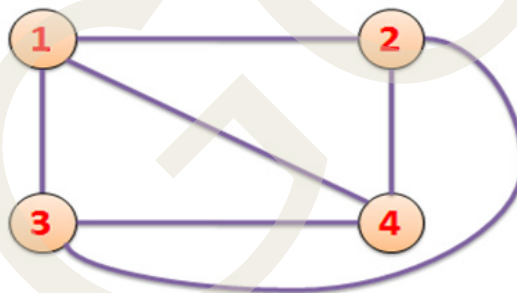


Fig 4.4.34 Example graph for multi list representation

We already know that the adjacency multi list representation is based on edges. The representation contains two parts; the directory of node information which consists of header nodes and the edge lists which consists of the information about the edges in the given graph. We can see these two parts in Figure 4.4.34.

The header nodes part is represented as an array structure. The nodes 1, 2, 3, and 4 are included in the header nodes section. Each node points to the corresponding edge list. There are 6 edges in this graph. So 6 edge lists are included in this representation. They are **N1**, **N2**, **N3**, **N4**, **N5**, and **N6**.

- ◆ The edge list starts with N1 which represents an edge (1, 2). So the header node 1 and header node 2 points to N1 because both the nodes are connected with this edge. In N1, the start vertex is 1 and end vertex is 2. The 4 th field

represents the first down list name where node 1 is present. Here it is N2 because the first down list which contains the node 1 is N2. (Down list of N1 is N2, N3, N4, N5, N6) The 5 th field represents the first down list name where node 2 is present. Here it is N4.

- ◆ The second edge list is N2. It represents the edge (1, 3). So the header node 3 points to N2. In N2, the start vertex is 1 and end vertex is 3. The 4 th field represents the first down list name where node 1 is present. Here it is N3 because the first down list which contains the node 1 is N3. (Down list of N2 is N3, N4, N5, N6) The 5 th field represents the first down list name where node 3 is present. Here it is N4.
- ◆ The third edge list is N3. It represents the edge (1, 4). So the header node 4 points to N3. In N3, the start vertex is 1 and end vertex is 4. The 4 th field represents the first down list name where node 1 is present. Here no down list contains the node 1. So we enter a zero or null value to that field. Here we enter a 0. (Down list of N3 is N4, N5, N6) The 5 th field represents the first down list name where node 4 is present. Here it is N5.
- ◆ The fourth edge list is N4. It represents the edge (2, 3). In N4, the start vertex is 2 and end vertex is 3. The 4 th field represents the first down list name where node 2 is present. Here it is N5 because the first down list which contains the node 2 is N5. (Down list of N4 is N5, N6) The 5 th field represents the first down list name where node 3 is present. Here it is N6.
- ◆ The fifth edge list is N5. It represents the edge (2, 4). In N5, the start vertex is 2 and end vertex is 4. The 4 th field represents the first down list name where node 2 is present. Here no down list contains the node 2. So we enter a zero or null value to that field. Here we enter a 0. (Down list of N5 is N6) The 5 th field represents the first down list name where node 4 is present. Here it is N6.
- ◆ The sixth edge list is N6. It represents the edge (3, 4). In N6, the start vertex is 3 and end vertex is 4. The 4 th field represents the first down list name where node 3 is present. Here no down list contains the node 3. So we enter a zero or null value to that field. Here we enter a 0. (There is no down list for N6) The 5 th field represents the first down list name where node 4 is present. Here no down list contains the node 4. So we enter a zero or null value to that field. Here we enter a 0.

From this we can conclude the representation of each vertex by the edge lists (Figure 4.4.35) as;

Here;

- ◆ N1 represents the edge (1, 2).
- ◆ N2 represents the edge (1, 3).
- ◆ N3 represents the edge (1, 4).
- ◆ N4 represents the edge (2, 3).
- ◆ N5 represents the edge (2, 4).

- ◆ N6 represents the edge (3, 4).

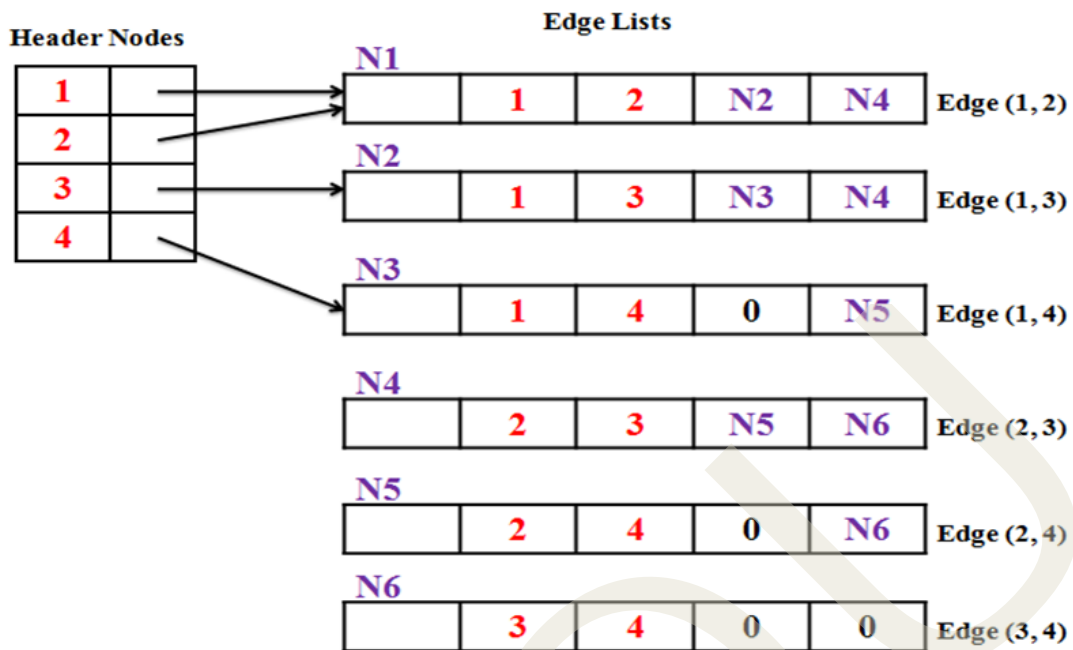


Fig 4.4.35 Multi list representation of example graph

- ◆ The edge list starts with N1 which represents an edge (1, 2). So the header node 1 and header node 2 points to N1 because both the nodes are connected with this edge. In N1, the start vertex is 1 and end vertex is 2. The 4th field represents the first down list name where node 1 is present. Here it is N2 because the first down list which contains the node 1 is N2. (Down list of N1 is N2, N3, N4, N5, N6) The 5th field represents the first down list name where node 2 is present. Here it is N4.
- ◆ The second edge list is N2. It represents the edge (1, 3). So the header node 3 points to N2. In N2, the start vertex is 1 and end vertex is 3. The 4th field represents the first down list name where node 1 is present. Here it is N3 because the first down list which contains the node 1 is N3. (Down list of N2 is N3, N4, N5, N6) The 5th field represents the first down list name where node 3 is present. Here it is N4.
- ◆ The third edge list is N3. It represents the edge (1, 4). So the header node 4 points to N3. In N3, the start vertex is 1 and end vertex is 4. The 4th field represents the first down list name where node 1 is present. Here no down list contains the node 1. So we enter a zero or null value to that field. Here we enter a 0. (Down list of N3 is N4, N5, N6) The 5th field represents the first down list name where node 4 is present. Here it is N5.
- ◆ The fourth edge list is N4. It represents the edge (2, 3). In N4, the start vertex is 2 and end vertex is 3. The 4th field represents the first down list name where node 2 is present. Here it is N5 because the first down list which contains the node 2 is N5. (Down list of N4 is N5, N6) The 5th field represents the first down list name where node 3 is present. Here it is N6.

- ◆ The fifth edge list is N5. It represents the edge (2, 4). In N5, the start vertex is 2 and end vertex is 4. The 4th field represents the first down list name where node 2 is present. Here no down list contains the node 2. So we enter a zero or null value to that field. Here we enter a 0. (Down list of N5 is N6) The 5th field represents the first down list name where node 4 is present. Here it is N6.
- ◆ The sixth edge list is N6. It represents the edge (3, 4). In N6, the start vertex is 3 and end vertex is 4. The 4th field represents the first down list name where node 3 is present. Here no down list contains the node 3. So we enter a zero or null value to that field. Here we enter a 0. (There is no down list for N6) The 5th field represents the first down list name where node 4 is present. Here no down list contains the node 4. So we enter a zero or null value to that field. Here we enter a 0.

From this we can conclude the representation of each vertex by the edge lists as;

- ◆ Vertex 1: N1 → N2 → N3
- ◆ Vertex 2: N1 → N4 → N5
- ◆ Vertex 3: N2 → N4 → N6
- ◆ Vertex 4: N3 → N5 → N6

4.4.6 Implementation of Graph

Consider a non weighted graph with the number of vertices MAX, where MAX is defined to 25. The basic node type which stores the information of a non weighted graph is defined in C as:

```
#define MAX 25      // defining the value of MAX to 25

typedef struct node  // creating a structure node
{
    int vertex;      // stores an integer value to the variable vertex
    struct node *next; // next will point to another structure of the type node
} node1; // declaring a structure variable

node1 *adj [MAX]; // declaring the adjacency list based on the number of vertices of graph
```

The above definition is called a structure definition. In this definition a name is given to the structure. The name of the above structure is node. That is we are creating a node using structure. This structure has two components. The first is of type int. We declare an integer variable vertex for storing the values of vertices or nodes. The second component is a structure of type nodes. Observe that we have defined a structure within a structure. This is allowed in C language. This declaration states that a pointer named next will point to another structure of the type node. That is, one vertex is pointing to the next one.

The use of typedef, names the structure as node1. That is we declare a structure variable node1 for further use of a node. Now we can declare the adjacency list on the basis of number of vertices of graph as node1 *adj [MAX]. That is we have declared adj[MAX] as a pointer which will point to data of type node1.

For a weighted graph the node structure is defined in C language as:

```
#define MAX 25 // defining the value of MAX to 25

typedef struct node // creating a structure node
{
    int vertex; // stores an integer value to the variable vertex
    int weight; // stores an integer value to the variable weight
    struct node *next; // next will point to another structure of the type node
}node2; // declaring a structure variable

node2 *adj [MAX]; // declaring the adjacency list based on the number of vertices of graph
```

The above definition is called a structure definition. In this definition a name is given to the structure. The name of the above structure is node. That is we are creating a node using structure. This structure has 3 components. The first and second are of type int. We declare an integer variable vertex for storing the values of vertices or nodes and an integer variable weight for storing the edge weights. The next component is a structure of type nodes. Observe that we have defined a structure within a structure. This is allowed in C language. This declaration states that a pointer named next will point to another structure of the type node. That is, one vertex is pointing to the next one. The use of typedef, names the structure as node2. That is we declare a structure variable node1 for further use of a node. Now we can declare the adjacency list on the basis of number of vertices of graph as node2 *adj [MAX]. That is we have declared adj[MAX] as a pointer which will point to data of type node2.

4.4.7 Graph Traversal

Traversal is a searching technique used in graphs. For searching a particular vertex in a given graph we use traversal. So the main goal of graph traversal is to find all vertices or nodes reachable from a given set of nodes. The order of vertices visited in a search process is also decided by the traversal. It also finds the edges to be used without creating any loops. That is in graph traversal all the vertices are visited without getting into a looping path. We can follow all the edges in an undirected graph. But in a directed graph we can follow only the out edges.

Traversal in graph is different from tree traversal because;

- ◆ There is no root node in the graph, so the traversal can start from any node.
- ◆ Only those nodes are traversed which are reachable from the starting node. If we want to traverse all the reachable nodes, we again have to select another

starting node for the remaining nodes.

- ◆ While traversing a graph there may be a possibility that we reach a node more than once. To ensure that each node is visited only once we have to keep the status of each node whether it has been visited or not.

The main graph traversal techniques are Breadth First Search (BFS) and Depth First Search (DFS). We can study each of them in detail in the following sessions. Breadth first search uses a queue structure for keeping the nodes and processing. In Depth first search we use a stack structure for node processing.

4.4.7.1 Breadth First Search (BFS)

BFS technique uses a queue for traversing all the nodes in the given graph. Here we first take any node as a starting node. That node is placed in the queue at first. So the front element of the queue is the starting node. Then we take all the adjacent nodes of the starting node. They are placed behind the starting node in the queue. After entering all the adjacent nodes of the starting node; that node is deleted from the queue. That is the first node is traversed successfully.

Then the second node in the queue becomes the front element of the queue. We take that node for processing. That is, we find all the adjacent nodes and place them in the queue if it is not present in the queue. So the front node in the queue is traversed successfully and we delete it from the queue. Then we take the next front node in the queue. A similar approach is done for all the other adjacent nodes placed in the queue. Likewise each node is processed.

Algorithm

Step 1. Define a queue for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and insert it into the queue.

Step 3. Delete the front element from the queue and insert all its unvisited adjacent nodes into the queue at the end.

Step 4. Repeat step 3 until the queue becomes empty.

Example:

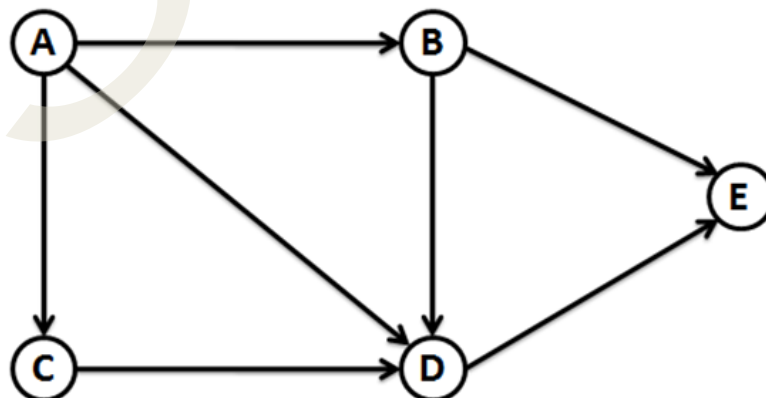


Fig 4.4.36 Example graph for BFS

Let us consider a graph shown in Figure 4.4.36 for breadth first traversing. Take the starting node as node A. The adjacent nodes of A are B, C, and D. The following steps illustrate the BFS.

1. Initially insert the starting node into the queue. That is insert node A into the queue. Here; $\text{Front} = \text{Rear} = 0$. The resultant graph and queue are shown in Figure 4.4.37.

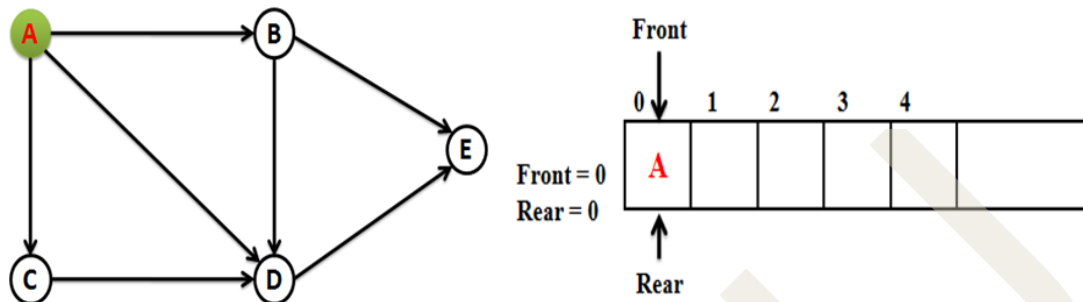


Fig 4.4.37 Breadth First Traversal - Resultant graph, queue (1)

2. Remove the front element A from the queue and increment $\text{Front} = \text{Front} + 1$. That is Front becomes 1. Insert the adjacent nodes of A to the queue if it is not present in the queue. Here; B, C, and D are the adjacent nodes and insert them into the queue.

So the $\text{Front} = 1$ and $\text{Rear} = 3$. Figure 4.4.38 shows the resultant graph and queue.

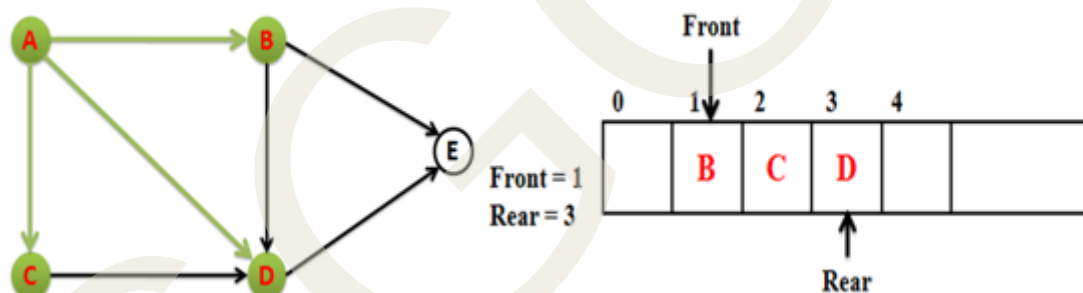


Fig 4.4.38 Breadth First Traversal - Resultant graph, queue (2)

3. Remove the front element B from the queue and add the adjacent nodes of B to the queue, if it is not present in the queue. The adjacent nodes of B are D and E. Here D is already in the queue. E is not in the queue. So insert E to the rear position. So the $\text{Rear} = 4$ and $\text{Front} = 2$. Figure 4.4.39 shows the resultant graph and queue.

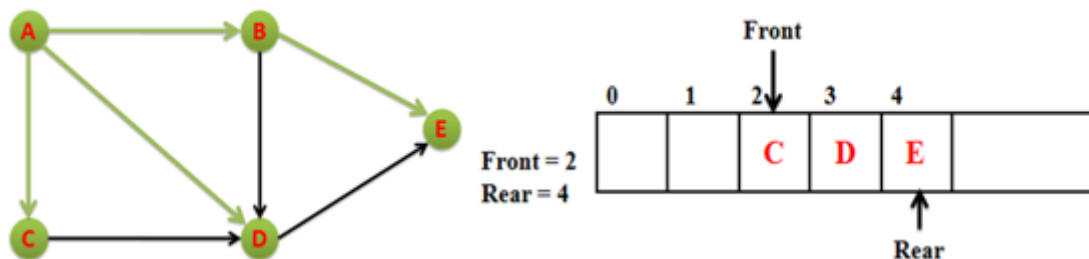


Fig 4.4.39 Breadth First Traversal - Resultant graph, queue (3)

4. Remove the front element C from the queue and add the adjacent nodes of C to the queue, if it is not present in the queue. The adjacent node of C is D. Here D is already in the queue. So the Rear = 4 and Front = 3. Figure 4.4.40 shows the resultant graph and queue.

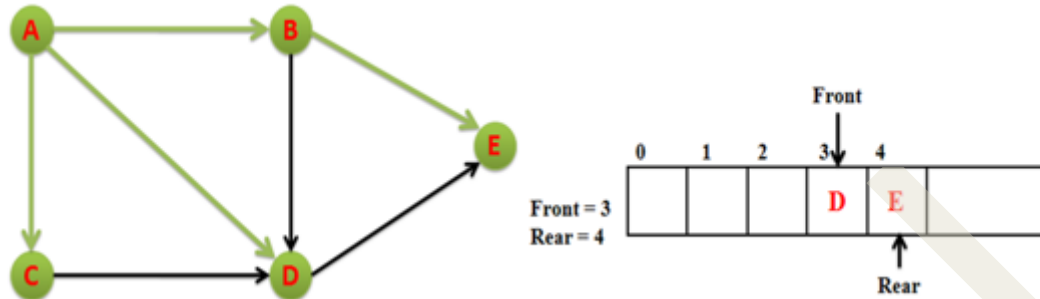


Fig 4.4.40 Breadth First Traversal - Resultant graph, queue (4)

5. Remove the front element D from the queue and add the adjacent nodes of D to the queue, if it is not present in the queue. The adjacent node of D is E. Here E is already in the queue. So the Rear = 4 and Front = 4. Figure 4.4.41 shows the resultant graph and queue.

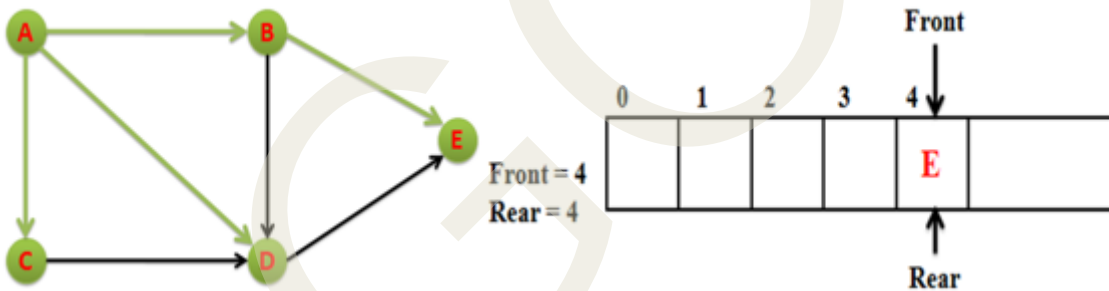


Fig 4.4.41 Breadth First Traversal - Resultant graph, queue (5)

6. The process is repeated until Front & Rear. Remove the front element E from the queue and add the adjacent nodes of E to the queue, if it is not present in the queue. Here the node E has no adjacent nodes. So the queue becomes empty. Also Front and Rear. Here Front = 5 and Rear = 4. So we stop the traversal. Figure 4.4.42 shows the resultant graph and queue.

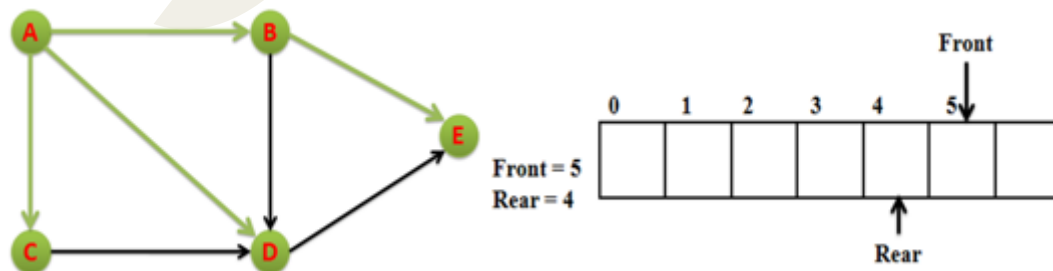


Fig 4.4.42 Breadth First Traversal - Resultant graph, queue (6)

And finally the traversed nodes are A, B, C, D, E.

4.4.7.2 Depth First Search (DFS)

Depth First Search (DFS) starts from a source node. We can select any node from the graph as a starting or source node. If S1 is the starting node, then DFS first visits S1 and then visits any one of its adjacent nodes, say S2. Then DFS again visits an adjacent node of S2, say S3. In the next step DFS again visits any one of the adjacent nodes of S3, say S4 and so on. DFS uses a stack for processing nodes. DFS uses a backtracking method for traversing all unvisited nodes in the graph. For example if node S4 has no adjacent nodes then it backtracks the traversal to the previous node. That is S4 backtracks to S3. Then DFS looks for any other adjacent node of S3. If it exists then visit that node. If not exists then backtrack to the previous node S2. This process is repeated until all the unvisited nodes get visited.

Algorithm:

Step 1. Define a stack for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and push it into the stack.

Step 3. Visit any one of the unvisited adjacent nodes of a node which is on the top of the stack and push it onto the stack

Step 4. Repeat Step 3 until there is no new node to be visited from the node which is on the top of the stack.

Step 5. If there is no unvisited adjacent node, it remains to backtrack the traversal and pop the top node from the stack.

Step 6. Again repeat the Steps 3, 4, and 5 until the stack becomes empty.

Example: Let us consider a graph shown in Figure 4.4.43 for depth first traversing. Take the starting node as node A.

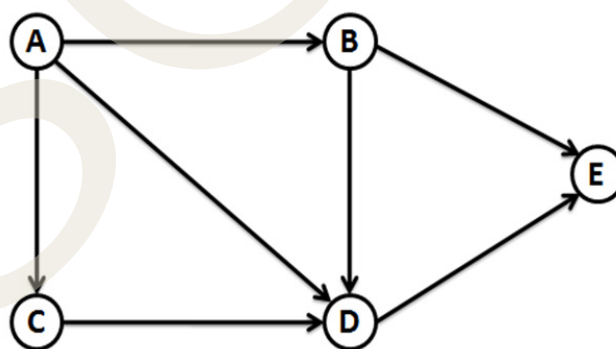


Fig 4.4.43 Example graph for DFS

The following steps illustrate the DFS.

1. Initially insert the starting node into the stack. That is push node A into the stack. Here top = 0. Figure 4.4.44 shows the resultant graph and stack.

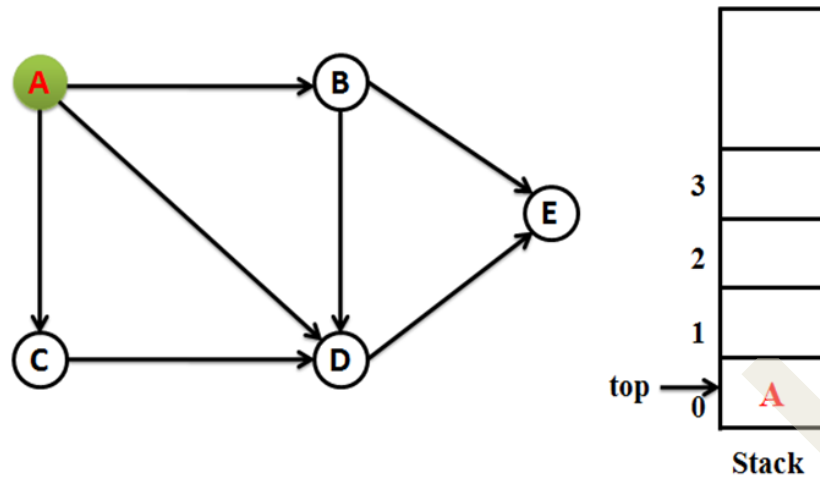


Fig 4.4.44 Depth First Traversal - Resultant graph, stack (1)

2. Visit any one of the unvisited adjacent nodes of A. Here visit node B. Then push B into the stack. So $\text{top} = 1$. Figure 4.4.45 shows the resultant graph and stack.

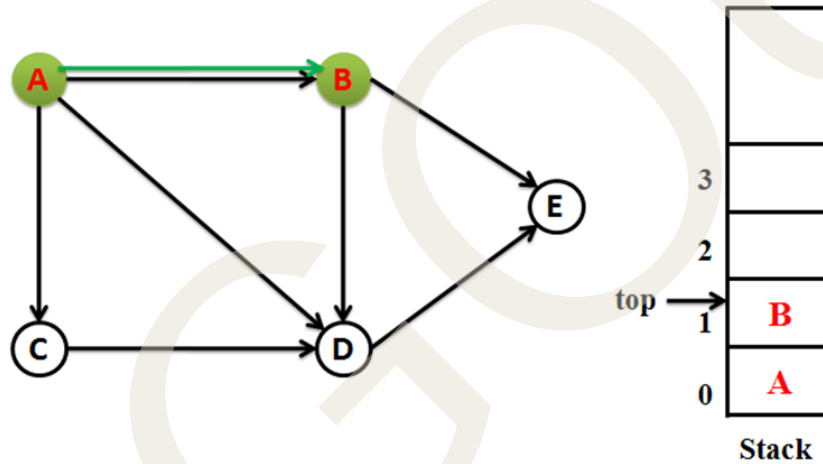


Fig 4.4.45 Depth First Traversal - Resultant graph, stack (2)

3. Visit any one of the unvisited adjacent nodes of B. Here we visit the node D. Then push D into the stack. So $\text{top} = 2$. Figure 4.4.46 shows the resultant graph and stack.

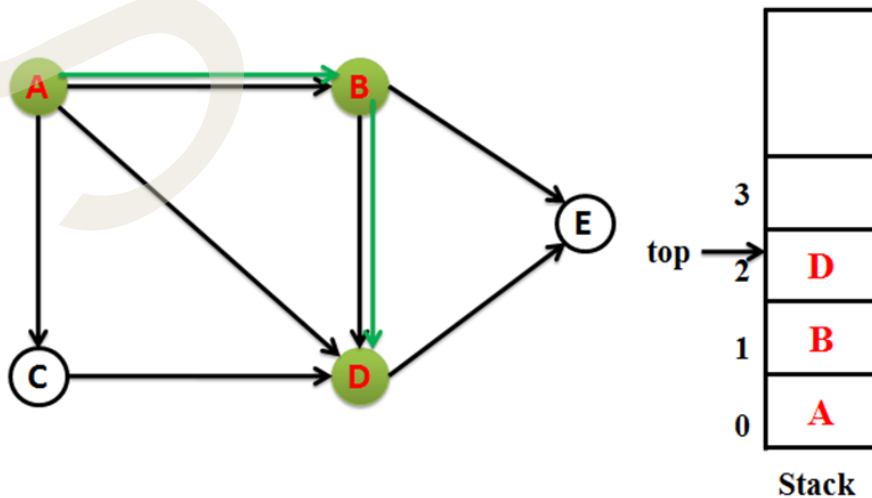


Fig 4.4.46 Depth First Traversal - Resultant graph, stack (3)

4. Visit any one of the unvisited adjacent nodes of D. Here the node is E. So we visit the node E. Then push E into the stack. So $\text{top} = 3$. Figure 4.4.47 shows the resultant graph and stack.

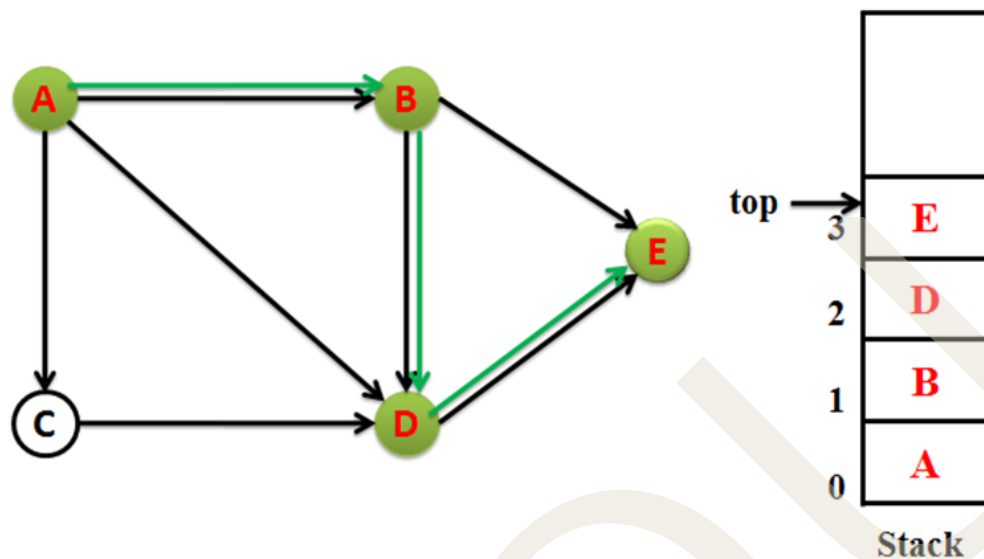


Fig 4.4.47 Depth First Traversal - Resultant graph, stack (4)

5. Visit any one of the unvisited adjacent nodes of E. There are no adjacent nodes for node E. So we backtrack the traversal. Pop node E from the stack. So $\text{top} = 2$. Figure 4.4.48 shows the resultant graph and stack.

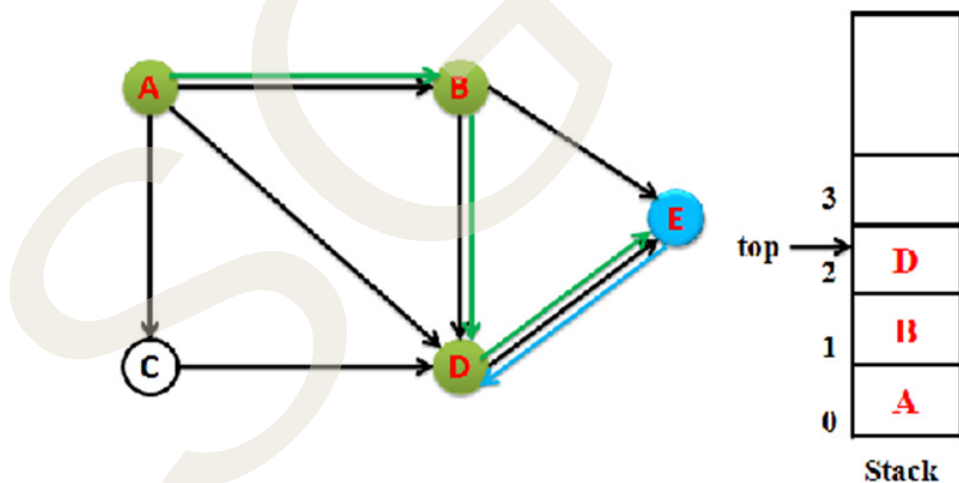


Fig 4.4.48 Depth First Traversal - Resultant graph, stack (5)

6. Visit any one of the unvisited adjacent nodes of D. There is no other unvisited adjacent node remaining for node D. So we backtrack the traversal. Pop node D from the stack. So $\text{top} = 1$. Figure 4.4.49 shows the resultant graph and stack.

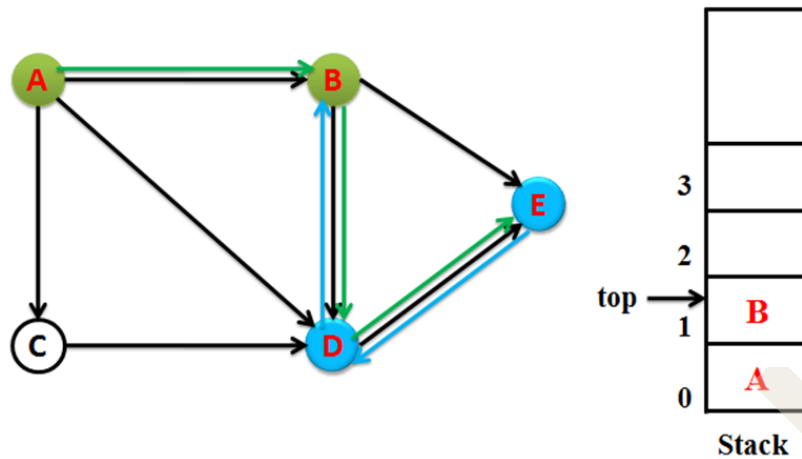


Fig 4.4.49 Depth First Traversal - Resultant graph, stack (6)

7. Visit any one of the unvisited adjacent nodes of B. There is no other unvisited adjacent node remaining for node B. So we backtrack the traversal. Pop node B from the stack. So $\text{top} = 0$. Figure 4.4.50 shows the resultant graph and stack.

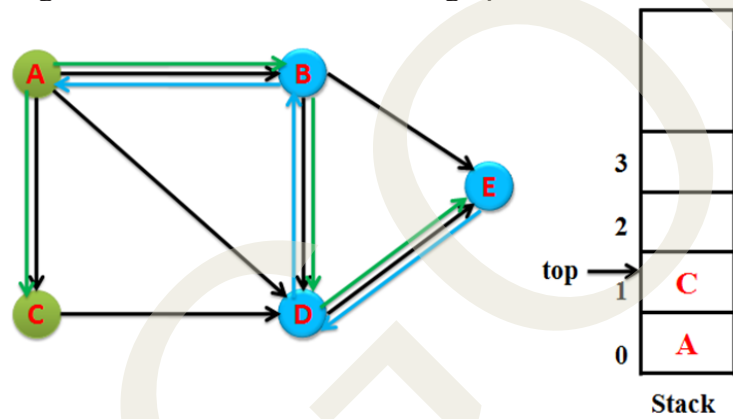


Fig 4.4.50 Depth First Traversal - Resultant graph, stack (7)

8. Visit any one of the unvisited adjacent node of A. Here only one unvisited node of A remains and that is the node C. So we visit the node C. Then push C into the stack. So $\text{top} = 1$. Figure 4.4.51 shows the resultant graph and stack.

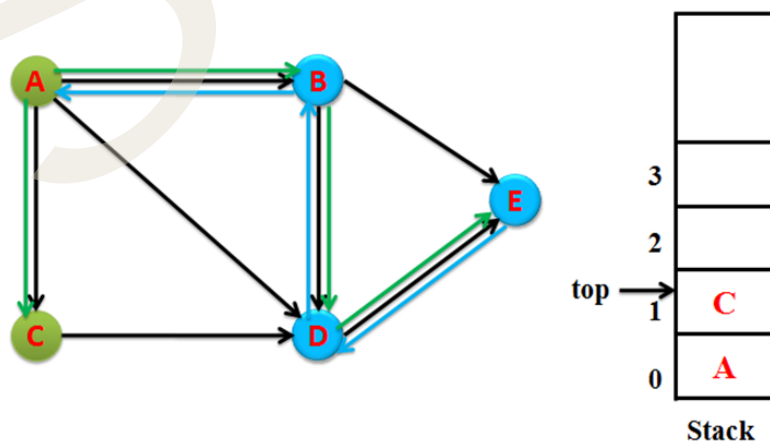


Fig 4.4.51 Depth First Traversal - Resultant graph, stack (8)

9. Visit any one of the unvisited adjacent nodes of C. There is no other unvisited adjacent node remains for node C. So we backtrack the traversal. Pop node C from the stack. So $top = 0$. Figure 4.4.52 shows the resultant graph and stack.

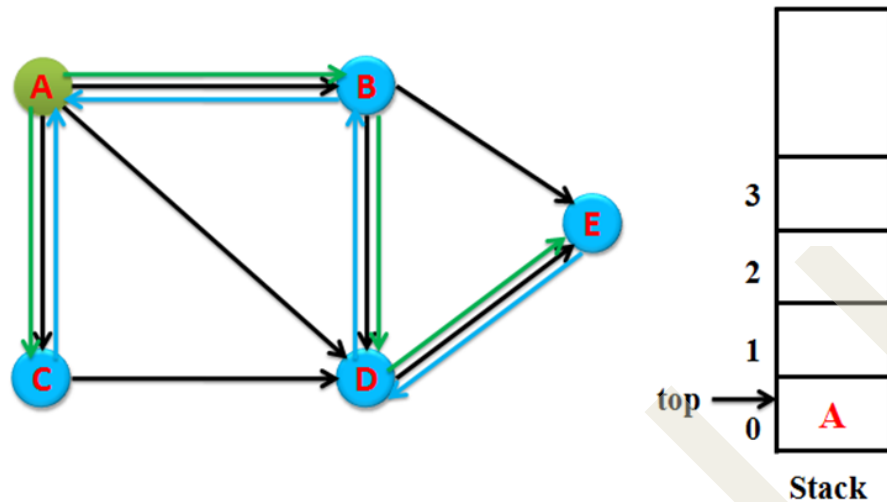


Fig 4.4.52 Depth First Traversal - Resultant graph, stack (9)

10. Visit any one of the unvisited adjacent nodes of A. There is no other unvisited adjacent node remaining for node A. So we backtrack the traversal. Pop node A from the stack. So $top = -1$. That is, the stack becomes empty. So we can stop DFS. Figure 4.4.53 shows the resultant graph and stack.

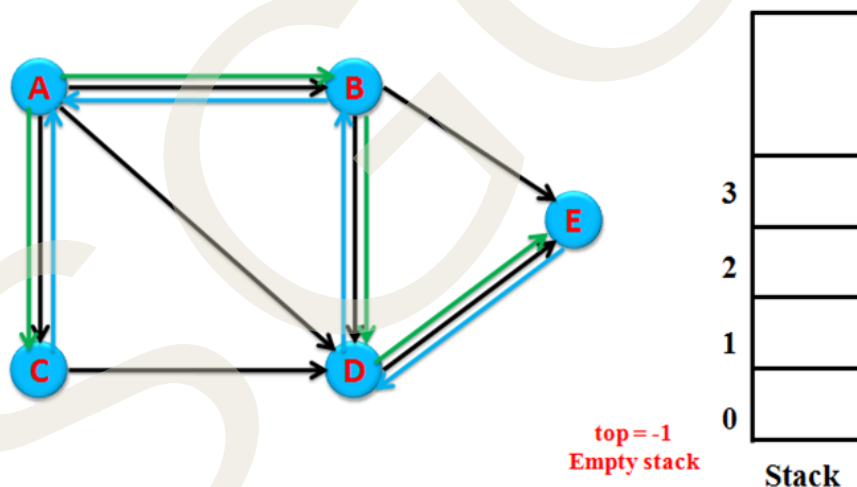


Fig 4.4.53 Depth First Traversal - Resultant graph, stack (10)

And finally all the nodes are traversed.

Note:

1. Suppose the searching item is very near to the starting vertex then BFS is used.
2. If the searching item is far from the starting vertex then DFS is used.
3. BFS is vertex based traversal.

4. DFS is edge based traversal.

4.4.8 Applications of Graph

Graphs are versatile data structures used to model relationships between entities, making them essential in various applications. Some of the important applications of graphs are traveling salesperson problem, GPS Navigation Systems, Social Networks, Knowledge Graphs, etc.

4.4.8.1 Traveling Salesman Problem (TSP)

Suppose you are an area manager of a financial firm. Your duty is to manage all the branches coming under your area. Your office is attached to the main branch. But you have to visit all the other branches periodically. Your higher authority insists that you make a quick inspection in all branches that come under you and submit the report to him in five days. What will you do first? How you will manage this situation. You have to find the shortest route to visit each branch and return back to your office without failure. A proper planning is needed for this purpose. The Traveling Salesman Problem (TSP) is also like this. TSP consists of a salesman and a set of cities. The salesman has to visit all the cities starting from a certain one (home town), by selecting shortest routes and returning back to his place where he started. So the challenge of TSP is to minimize the total length of the trip.

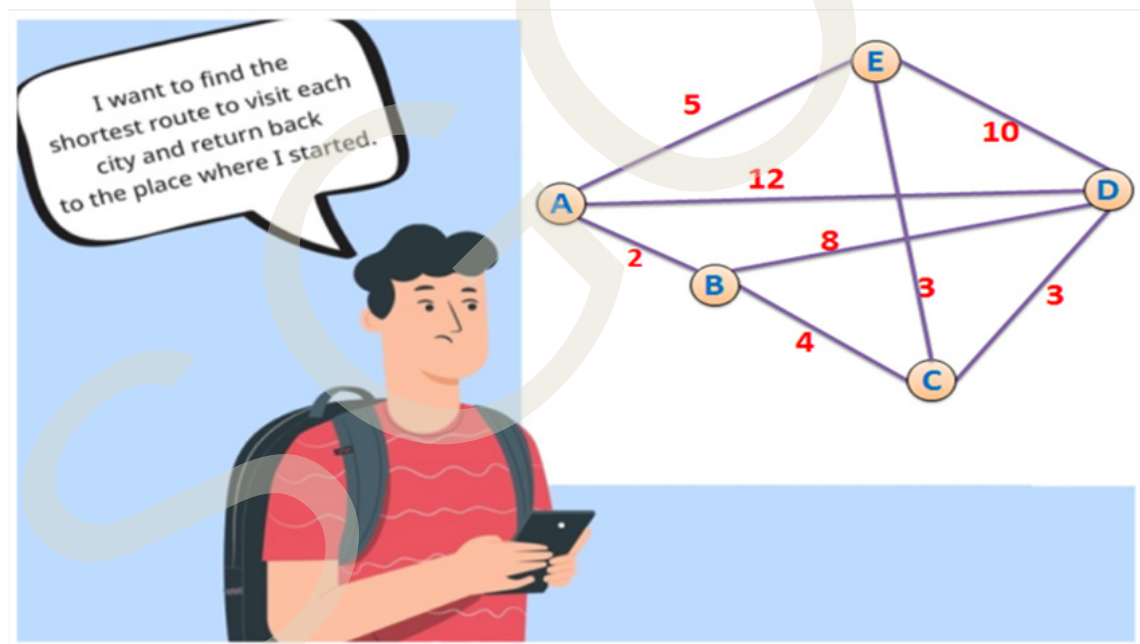


Fig 4.4.54 Traveling Salesman Problem

In Figure 4.4.54, we can see a set of cities represented by a graph. Here each city is represented by a node. Here the problem lies in finding a minimal path passing from all nodes once. Suppose the home town is A. That is the salesman starts his journey from the city A. He must visit all the cities (B, C, D, and E) and returns back to his home town (A). So he has to find a route with minimum distance that starts from A and pass through all the cities once and return back to A. For example take the path

A-B-C-D-E-A. We can represent it as Path 1 = {A, B, C, D, E, A}. And take the path A-B-C-E-D-A. We can represent it as Path 2 = {A, B, C, E, D, A}. Both the paths pass all the nodes but Path 1 has a total length of 24 and Path 2 has a total length of 31. So considering all other possible paths, the salesman has to choose a shortest route. Here Path 1 is the shortest route and it is selected.

4.4.8.2 Google Map

Suppose you are going to attend your friend's marriage at Kunnamkulam. But you have no idea about that place. Now you are staying at Wadakkanchery. What will you do? You will search that place in Google map and you can see (Figure 4.4.55) the different routes to Kunnamkulam from Wadakkanchery. It shows you the distance and approximate travel time to reach the destination.

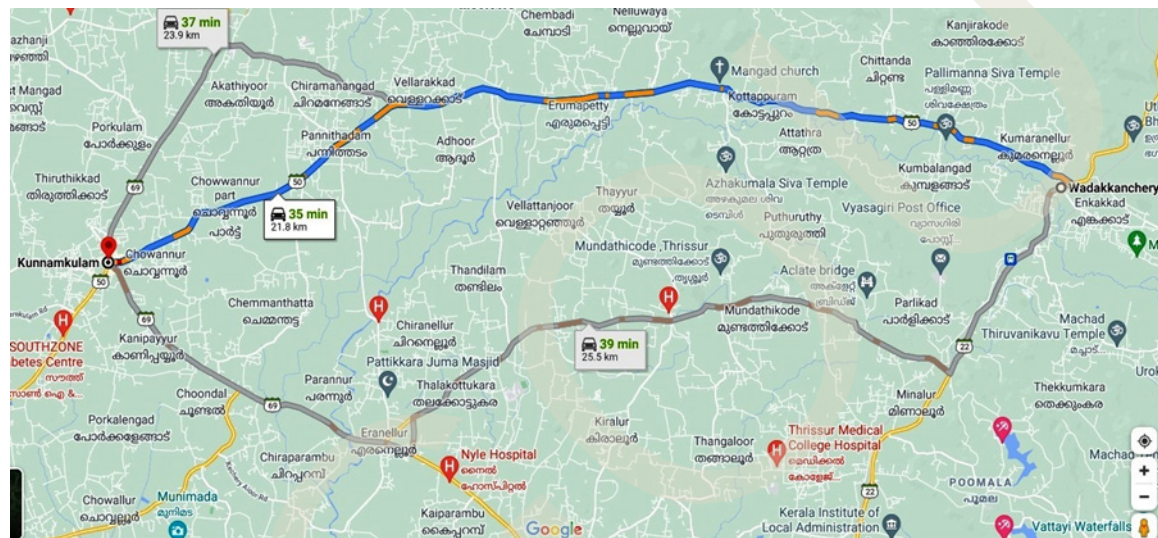


Fig 4.4.55 Google Map

Google maps uses graphs for building transportation systems. In Google map, various locations are represented as vertices and roads connecting these locations are represented as edges. The intersection of two or more roads is also considered to be a vertex. The navigation system is based on the algorithm to calculate the shortest path from one vertex to another. In the previous example we can see three paths between Wadakkanchery and Kunnamkulam. We can represent these paths using graphs.

Figure 4.4.56 shows the graph representation. Here Wadakkanchery is the source vertex represented with vertex 1 and Kunnamkulam is the destination vertex represented with vertex 6. The interconnection of two or more roads that are coming in these different routes (paths) can be represented as vertices (vertices 2, 3, 4 and 5). Here w_{12} is the distance between nodes 1 and 2 and w_{26} is the distance between the nodes 2 and 6.

Likewise w_{13} , w_{35} , w_{56} , w_{24} and w_{46} are the distances between the corresponding nodes. Here three paths exist between node 1 and node 6. They are; Path 1= 1-2-6, Path 2= 1-2-4-6 and Path 3= 1-3-5-6. Here the shortest path is path 1. So it is recommended to the user.

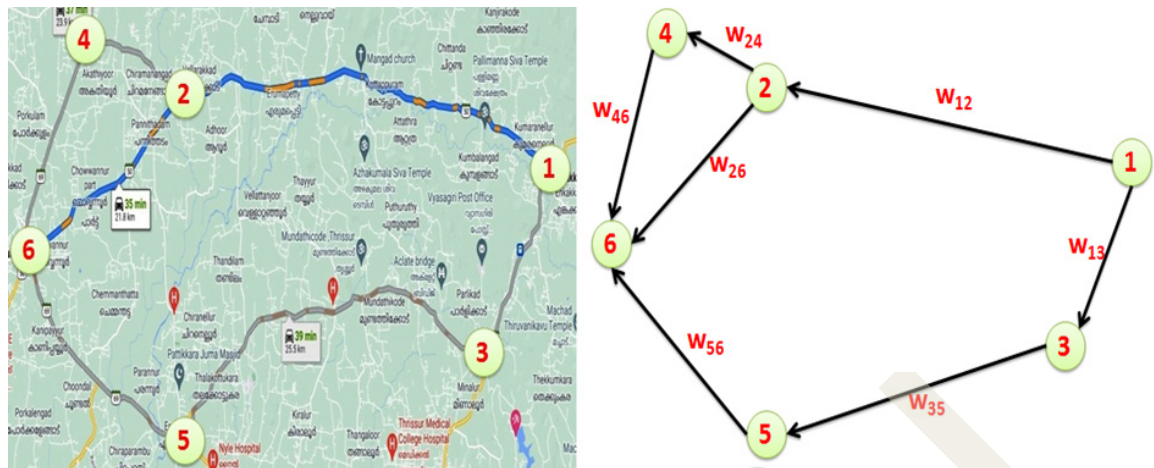


Fig 4.4.56 Graph representation of routes in Google Map

4.4.8.3 GPS Navigation System

For vehicle navigation we use the Global Positioning System or GPS. It is also a type of shortest path routing API and differs from Google Maps routing API because it uses a single source (from one vertex to every other). That is it computes locations from where you are to any other location you might be interested in going. Here BFS is used to find all the neighboring locations. Stand alone GPS devices actually store their own map data compared to a smart phone which is cloud based. It requires a connection to download the maps as you go.



Fig 4.4.57 Examples for GPS Navigation Systems

If you don't have a connection, whether it be 3G, 4G or Wifi, you will get a blank screen. Figure 4.4.57 shows examples (MAPMYINDIA, Primo GPS etc.) for GPS navigation systems.

4.4.8.4 Flight Networks

For flight networks, an efficient route optimization is needed. The graph data structure is perfectly fit for this purpose. Using graph models, airport procedures can be modeled and optimized efficiently. Graphs are used to compute shortest paths and fuel usage in route planning. The vertices in flight networks are places of departure and destination, airports, aircrafts, cargo weights etc.

The flight trajectories between airports are the edges. Entities such as flights can have properties such as fuel usage, crew pairing which can themselves be more graphs. Figure 4.4.58 shows a simple flight network graph of a particular Airline. It includes airports as vertices and flights between the airports as edges.

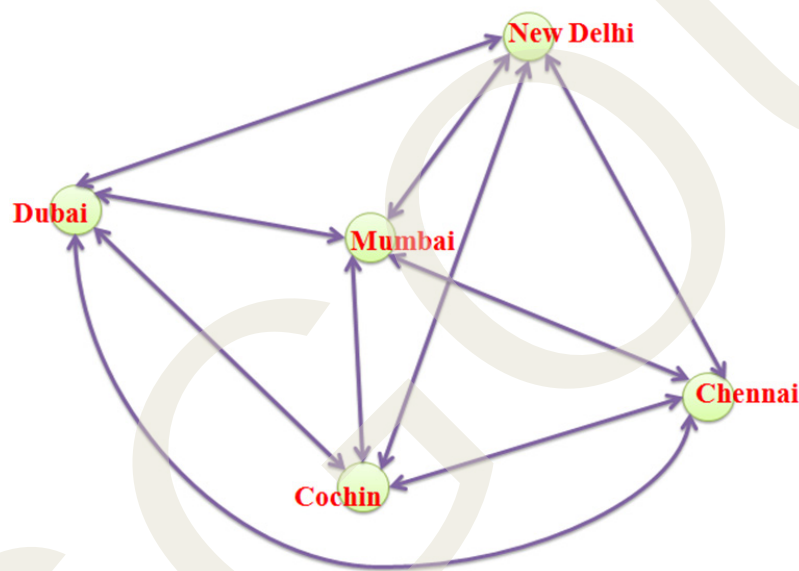


Fig 4.4.58 A simple flight network graph

From Figure 4.4.58, we can see the Airline operates between 5 five major airports. A particular flight from Cochin to Mumbai is represented as a directed edge from vertex Cochin to the vertex Mumbai.

4.4.8.5 Social Networks

Suppose you are browsing on the Internet and lots of web pages are loaded to your browser according to your browsing. When you click on a link, it will redirect you to some other page. The graph data structure plays a significant role in the World Wide Web also. In WWW, web pages are considered to be the vertices. If a link from a page u to a page v is there, then there will be an edge from the page u to the page v .

The best example of application of graphs in our real life is the social networks like Facebook, Twitter etc. The entities like Users, Pages, Places, Groups, Comments, Photos, Photo Albums, Stories, Videos, Notes, and Events in social networks are represented

as vertices or nodes in graphs. Anything that has properties that store data is a vertex. Every connection or relationship is an edge. For example, a user posting a photo, video or comments etc. all are represented with edges. A user updating his profile with place of birth, a relationship status is also represented by edges. Suppose you like a photo of your friend, an edge is inserted between you and that photo.

A real social network would have millions and billions of nodes. Figure 4.4.59 shows only a few nodes of a social network. A social network like Facebook can be represented as an undirected graph. Here each user is represented as a vertex or node. If two users are friends then there would be an edge connecting between them. Friendship is a mutual relationship. If I am your friend, then you are my friend too. So connections have to be two way. That is Facebook is an undirected graph. In Figure 4.4.59, we can see the friendship between the users is represented with a graph. Here each node is a user.

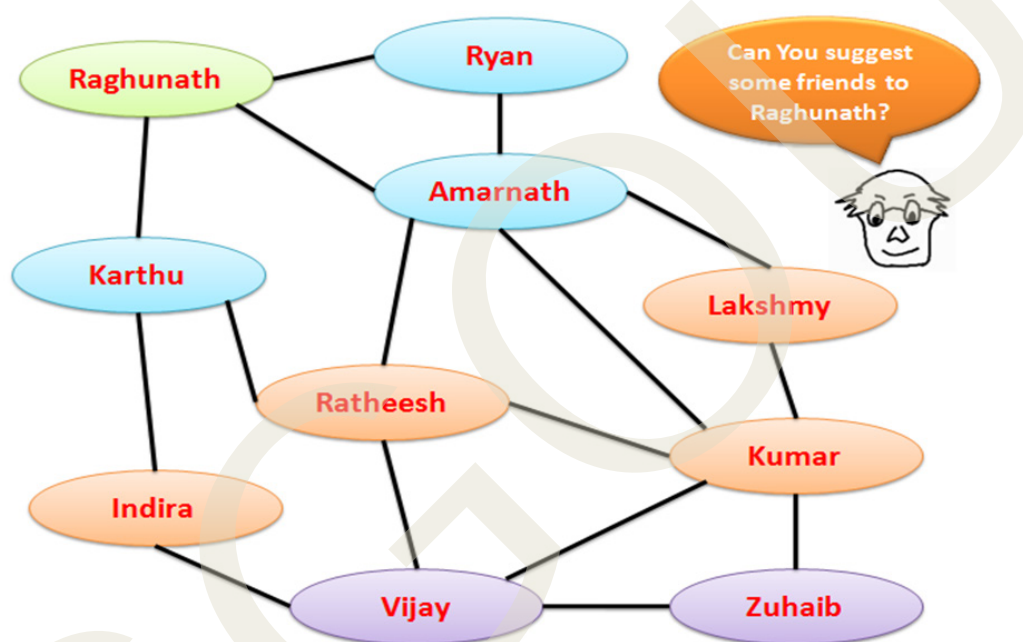


Fig 4.4.59 Graphs in Social Network

Now we want to do something like suggest friends to a user. It is a common thing in social networks. Let's say we want to suggest some friends to Raghunath (Figure 4.4.59). One possible approach to do so can be suggesting friends of friends who are not connected already. Raghunath has 3 friends; Amarnath, Ryan and Karthu and the friends of these three that are not connected to Raghunath already can be suggested. There is no friend of Ryan who is not connected to Raghunath already. Amarnath however has 3 friends Lakshmy, Kumar and Ratheesh that are not friends with Raghunath. So they can be suggested. And Karthu has 2 friends; Ratheesh and Indira that are not connected to Raghunath. We have counted Ratheesh already. So in all we can suggest these 4 users (friends) to Raghunath. We described this problem (suggesting friends to a user) in the context of a social network. This is a standard graph problem. The problem here in pure graph terms is finding all nodes having length of shortest path from a given node equal to 2. In this example, the length of the shortest path from Raghunath to Indira, Ratheesh, Kumar and Lakshmy is 2. Twitter is an example of a very large scale

complex graph. Here users and tweets are represented as vertices. The interactions such as follow, replies, likes, posts and retweets are represented with edges.

4.4.8.6 Recommendation On E-Commerce Websites

We all are familiar with e-commerce websites like Amazon, Flipkart etc. Suppose you want to buy a LED TV from one of these e-commerce websites. When you search for a product on this website, you can notice various options about things you want to buy or are interested in. E-commerce websites use a technology called recommendation systems. Recommendation systems track your profile information like what kinds of products you buy, which pages you click on, what products you are interested in etc. Based on your profile data, a recommendation system analyzes these data and provides you the recommendations.

So everyone using these e-commerce websites would receive individual personalized recommendations based on their browsing patterns, purchase history etc. One way of storing these data is to use a graph data structure.



Fig 4.4.60 Example of User-Product graph.

Here all users and products are stored in nodes on a graph with edges connecting related data. That is if a user purchased a product then the user node and the product node get connected with an edge. Consider an example shown in Figure 4.4.60. Here Raghunath is a user and he wants to buy a LED TV. Both the user (Raghunath) and the product (LED TV) are represented by nodes. After purchasing the product an edge connects both the nodes in the graph representation. Here the graph shows that Raghunath purchased a LED TV.

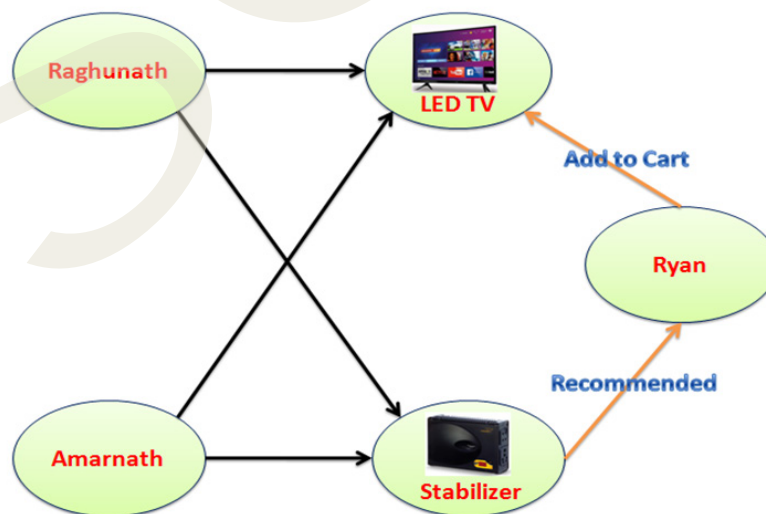


Fig 4.4.61 Example of Recommendation of products using graphs

We can use the structure of graphs to make recommendations very efficiently. The “Recommendations for you” section on various e-commerce websites uses graph theory to recommend items of similar type to the user’s choice. Figure 4.4.61 shows how the recommendation section works. When Ryan adds the LED TV to his cart, the recommendation system looks at all the users connected to the LED TV. In this example, Raghunath and Amarnath are connected to the LED TV. We can find other product nodes that are connected to both Raghunath and Amarnath, in this example the Stabilizer. We then recommend the Stabilizer to Ryan. The graph structure allows you to make these types of requests very quickly.

4.4.4 Knowledge Graphs

Suppose you have a tour plan to visit Paris. You booked your seat in a tour package named Tour Eiffel which is located in Paris. Your friend is also interested in this package because she wants to see the painting “Mona Lisa” by “Da Vinci”. But he didn’t get a seat in the Tour Eiffel Package because the booking was closed early. You visited the Louvre museum and saw the painting “Mona Lisa”. After your pleasure trip you returned to your home. After reaching your home you want to document your pleasure trip from starting to end. You have to convey lots of information from the beginning to the end of the journey. One way is to represent these in a pictorial format. Figure 4.4.62 shows a graphical representation of the context.

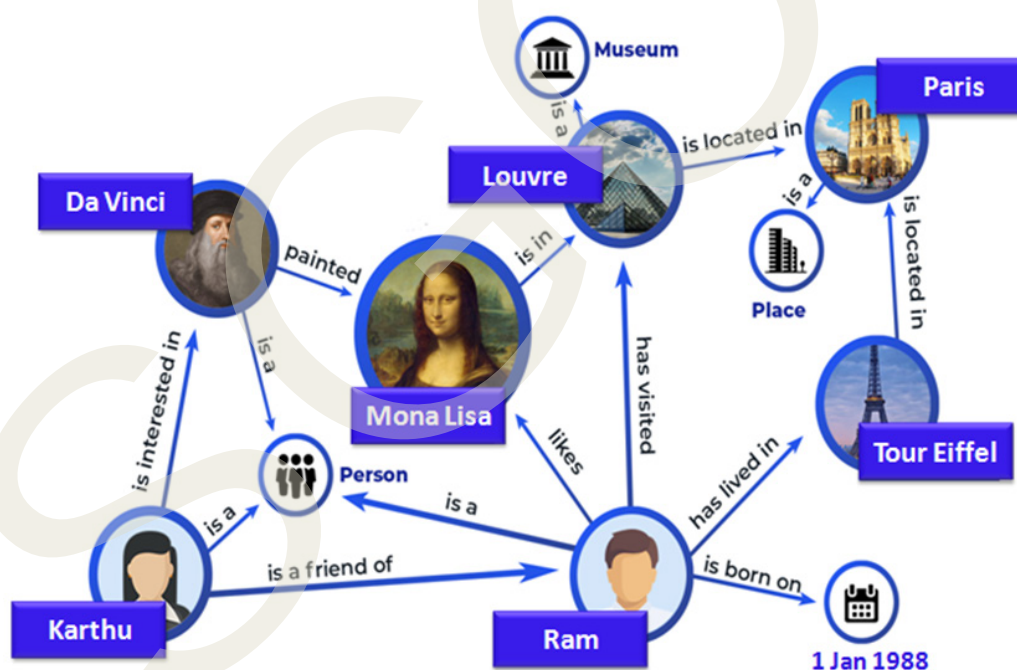


Fig 4.4.62 Example for Knowledge graph

Here the entities, objects, events etc. are represented by nodes and the relationship between the nodes is represented with edges. It includes the following information. Karthu (Your friend) is a friend of Ram (You). Both Karthu and Ram are people. Karthu is interested in Da Vinci and he is also a person. Da Vinci painted the Mona Lisa and it is in the Louvre museum. Ram likes Mona Lisa. Ram was born on 1 st January 1988.

Ram has lived in the Tour Eiffel package. It is located in Paris. Paris is a place. Ram has visited the Louvre museum which is located in Paris. So this graph structure integrates data to model a knowledge base. Knowledge base consists of lots of information. This type of representation shown in Figure 4.4.63 is known as knowledge graph.

Let's have a look at another interesting example related to the film industry. You may be familiar with Hollywood movies like "Jurassic Park", "Indiana Jones and the Kingdom of the Crystal Skull", and "War of the worlds" which are directed by the famous Hollywood director Steven Spielberg. All the 3 movies are Sci-Fi movies. Jurassic Park was released on June 11 th , 1993. War of the Worlds was released on June 29 th , 2005 and Indiana Jones and The Kingdom of The Crystal Skull was released on May 22 nd , 2008. We can represent this context using graphs.

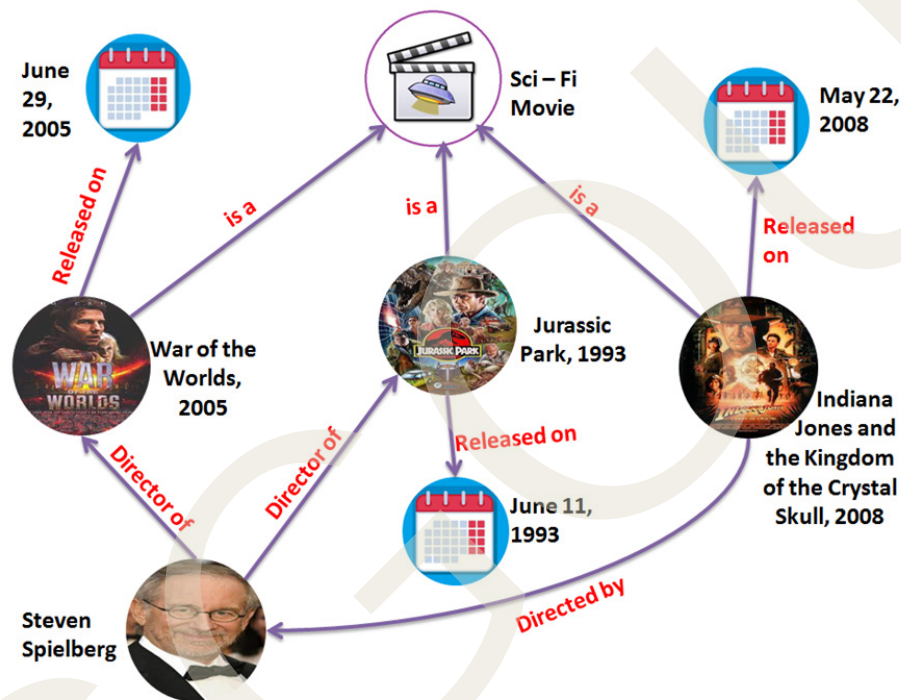


Fig 4.4.63 Knowledge representation using Graphs

Figure 4.4.63 shows the knowledge representation of this context by using graphs. Here the information is represented with nodes and links. Each link represents the relationship between the nodes. For example "Steven Spielberg is the director of Jurassic Park.", can be represented as;

Here in (1), "Steven Spielberg" and "Jurassic Park" are the two nodes. The relationship between the nodes is "Director of" and is represented on the link. Likewise all sentences are represented using nodes and links and combined to form a full representation of that particular context which is shown in Figure 4.4.63.

Thus a knowledge graph represents a collection of interlinked descriptions of entities, objects, events or concepts. These entities, objects, events or concepts are represented as nodes and the relationship between them is represented with the edges. So a knowledge graph is a knowledge base that uses a graph structured data model to integrate data.

In summary, the study of graphs forms a vital part of data structures and algorithms, offering powerful tools for modeling and solving real-world problems involving relationships and connections. By understanding the different types of graphs and the essential terminologies, learners gain the ability to describe and analyze graph-based structures effectively. Knowledge of graph representation techniques, such as adjacency matrices and lists, enables efficient storage and manipulation of graph data. Furthermore, mastering graph traversal methods like Depth First Search (DFS) and Breadth First Search (BFS) equips students with the skills to explore graphs systematically, which is fundamental for tasks like searching, pathfinding, and network analysis. Together, these concepts provide a strong foundation for advanced topics in computer science and practical applications in various domains.

Recap

- ◆ A graph is a collection of vertices (nodes) connected by edges.
- ◆ Graphs can be directed (edges have direction) or undirected (edges have no direction).
- ◆ A weighted graph assigns a weight or cost to each edge, while an unweighted graph does not.
- ◆ A simple graph has no loops or multiple edges between the same vertices.
- ◆ A cyclic graph contains at least one cycle, while an acyclic graph contains none.
- ◆ The degree of a vertex is the number of edges connected to it; in directed graphs, degree is divided into in-degree and out-degree.
- ◆ An adjacent vertex is a vertex connected directly by an edge.
- ◆ A path is a sequence of edges connecting a sequence of vertices.
- ◆ A connected graph has a path between every pair of vertices; otherwise, it is disconnected.
- ◆ Graphs can be represented mainly in two ways: adjacency matrix and adjacency list.
- ◆ The adjacency matrix is a 2D array where each element indicates if an edge exists between vertex pairs.
- ◆ The adjacency list stores, for each vertex, a list of its adjacent vertices.
- ◆ The adjacency list is more efficient for sparse graphs, while adjacency matrix is better for dense graphs.
- ◆ Graph traversal is the process of visiting all the nodes in a graph in a systematic way.

- ◆ Depth First Search (DFS) explores as far along a branch as possible before backtracking.
- ◆ DFS uses a stack data structure or recursion to keep track of vertices to visit.
- ◆ Breadth First Search (BFS) explores all neighbors of a vertex before moving to the next level.
- ◆ BFS uses a queue data structure to explore vertices in layers.
- ◆ DFS is useful for detecting cycles and finding connected components.
- ◆ BFS is commonly used for finding the shortest path in an unweighted graph.

Objective Type Questions

1. What does a vertex in a graph represent?
2. In a directed graph, edges are called what?
3. What type of graph has edges with no direction?
4. What term describes the number of edges connected to a vertex?
5. In a directed graph, what is the number of edges coming into a vertex called?
6. What is a graph called if it contains no cycles?
7. What data structure is commonly used to implement DFS?
8. What data structure is used to implement BFS?
9. Which graph representation uses a 2D matrix to store edges?
10. Which graph representation uses lists to store adjacent vertices?
11. Which graph representation is more space-efficient for sparse graphs?
12. BFS traversal visits nodes in what order?
13. DFS traversal visits nodes in what order?
14. What type of graph traversal is best for finding the shortest path in an unweighted graph?
15. What does a weighted graph have that an unweighted graph does not?
16. A graph with no edges is called?
17. What does connectivity in a graph mean?

18. What is a cycle in a graph?
19. In adjacency matrix, what value represents the absence of an edge?
20. What traversal method can be implemented recursively?

Answers to Objective Type Questions

1. A node or point in the graph.
2. Arcs.
3. Undirected graph.
4. Degree.
5. In-degree.
6. Acyclic graph.
7. Stack.
8. Queue.
9. Adjacency matrix.
10. Adjacency list.
11. Adjacency list.
12. Level by level (breadth-wise).
13. Depth-wise (goes deep before backtracking).
14. BFS (Breadth First Search).
15. Weights or costs associated with edges.
16. Null graph or empty graph.
17. There is a path between every pair of vertices.
18. A path that starts and ends at the same vertex without repeating edges.
19. 0 (zero).
20. DFS (Depth First Search).

Assignments

1. Explain the different types of graphs and their characteristics. Provide examples for each type.
2. Describe the common terminologies used in graph theory. Illustrate each term with diagrams where appropriate.
3. Compare and contrast adjacency matrix and adjacency list graph representations. Discuss their advantages, disadvantages, and scenarios where each is preferred.
4. Explain the algorithms for Depth First Search (DFS) and Breadth First Search (BFS) graph traversal. Provide step-by-step examples for both traversals on a sample graph.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
4. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

Suggested Reading

1. GeeksforGeeks. (n.d.). Graph Data Structure and Algorithms. <https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>
2. Tutorialspoint. (n.d.). Graph Data Structure. https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.html
3. Programiz. (n.d.). Graph Data Structure and Algorithms. <https://www.programiz.com/dsa/graph>
4. Brilliant.org. (n.d.). Graphs and Graph Algorithms. <https://brilliant.org/wiki/graph-algorithms/>
5. Coursera – Algorithms Specialization by Stanford University. <https://www.coursera.org/specializations/algorithms>



Design Algorithms



Unit 1

Complexity of Algorithms

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ explain the properties of an algorithm.
- ◆ understand different complexities of algorithms based on input size.
- ◆ familiarize different asymptotic notations in algorithm designing.

Prerequisites

Have you ever wondered why some computer programs run in the blink of an eye while others seem to take forever to finish the same kind of task? This mystery lies in something called algorithmic complexity. The hidden measure of how efficiently a program uses time and memory. As computers handle larger and larger amounts of data, understanding this concept becomes crucial. Even a tiny difference in the way an algorithm is designed can mean the difference between finishing a task in seconds or waiting hours for the result.

Learning about the complexity of algorithms allows you to see beyond the code to think like a problem-solver and an optimizer. It helps you compare multiple solutions to a problem and choose the one that performs best under real-world conditions. In the world of technology, where speed and efficiency often define success, this knowledge gives you a competitive edge. Whether you're designing a search engine, sorting millions of records, or developing an AI model, understanding complexity helps you build systems that are not just correct, but cleverly efficient.

Moreover, this topic shapes your analytical thinking. It teaches you to estimate how an algorithm will behave as the input grows and to recognize potential bottlenecks even before running the program. In essence, learning about algorithmic complexity is like learning to see the “energy cost” of your code — empowering you to write smarter, faster, and more sustainable software for the future.

Key concepts

Time complexity, space complexity, Big-O, Big-Ω, Big-Θ



Discussion

5.1.1 Essential Properties of algorithms

A finite set of instructions that accomplishes a particular task is known as an algorithm. Following are the properties of an algorithm:

1. **Input:** An algorithm has zero or more inputs. These are the data items that are given to the algorithm initially before it starts executing.
2. **Output :** After executing an algorithm, we must get at least one output. This output is the result of the algorithm's processing of the input.
3. **Definiteness :** Each step of the instructions must be clear and unambiguous. This property of an instruction is known as definiteness.
4. **Finiteness :** Finiteness refers to the property that the algorithm terminates after a finite number of steps.
5. **Effectiveness:** Effectiveness refers to the property that every instruction must be feasible and that each instruction should do some task.

5.1.2 Cases to consider during analysis

Let us take the pizza baking example. Let us consider your mother and your friend's mother who are going to prepare pizza. Suppose your mother is very systematic and keeps every ingredient ready before preparing the pizza. In the case of your mother, the performance will be the best if we have all the ingredients ready and everything well arranged, as it will be easy for the preparation. Let us imagine that your friend's mother is not very systematic. In this case, even if we have all the ingredients ready and everything well arranged, she might not be able to perform well. However, there might be a cooking style that she is used to and she can prepare well. This situation might not be favorable for your mother. i. e the preparation is highly dependent on the kind of input(here ingredients) that you have provided.

In the same manner, the performance of an algorithm has significant impact based on the inputs considered when analyzing an algorithm. For example, if an input list is almost in the sorted order, some sorting algorithms will perform well, while some other sorting algorithms will perform poorly. But, the opposite would be the condition if the list is randomly arranged instead of sorted. Hence, while analyzing an algorithm, multiple input sets must be considered.

Following are the cases to be considered during analysis:

1. Best Case Input
2. Worst Case Input
3. Average Case Input

5.1.2.1 Best case Input

Let us take the example of your mother preparing the pizza where all the ingredients are ready and kept in an orderly fashion. Suppose she wanted to search for salt, since they are kept in an orderly fashion, she will get it easily. In this case, she will take the shortest amount of time to prepare.

Now, let us consider an example of an algorithm for searching a number from a group of numbers. If the number to be searched is found as the first number in the list, then the least amount of time is taken for the search. Such an input is known as best case input. i.e best case input represents the input set that allows the algorithm to perform quickest. This is because input takes the shortest time to execute, as it causes the algorithm to do the least amount of work.

5.1.2.2 Worst case Input

Let us take the example of your mother preparing the pizza where all the ingredients are not ready and are not kept in an orderly fashion. Suppose she wanted to search for salt, since they are not kept in an orderly fashion, she will not get it easily, and she will have to search the entire kitchen for the same. In this case, she will take the maximum amount of time to prepare.

Now, let us again consider the example of an algorithm for searching a number from a group of numbers. If the number to be searched is found as the last number in the list, then the maximum amount of time is taken for the search. Such an input is known as worst case input. i.e worst case input represents the input set that allows the algorithm to perform slowest. This is because input takes maximum time to execute, as it causes the algorithm to do the maximum amount of work.

5.1.2.3 Average case Input

Let us take the example of your mother preparing the pizza where some of the ingredients are available and some are not. In this case, she will take some more time than compared to her actual speed as she might have to ask you to purchase some ingredients, arrange ingredients whatever is available, etc. In this case, she will take an average amount of time to prepare.

In the same manner, average case input represents the input set that allows an algorithm to deliver an average performance.

5.1.3 Complexity of Algorithms

There must be some criteria to measure the efficiency of the algorithm. Measuring the efficiency of algorithms helps in comparing the algorithms. The time and space used by the algorithm are the two main measures of efficiency of the algorithm.

5.1.3.1 Time complexity

The time required to complete a task is crucial. For instance, in cooking, if all the ingredients are prepared in advance and the mother uses a simple cooking method, the time needed will be minimal. However, if she doesn't organize the ingredients and follows a more complex method, the cooking time will be significantly longer.

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

5.1.3.2 Space complexity

The correct amount of ingredients in cooking is crucial. Similarly, in our algorithm, the memory usage by each variable is very important.

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.

5.1.4 Estimating complexity

5.1.4.1 Time for an algorithm to run $T(n)$

Let us consider a situation where you have some numbers that are not in a sorted order. Let us consider the following numbers in random order.

9 4 6 2 5 3

In order to sort the above list, there are various techniques. Before we consider the cases, let us define the numbers in terms of n .

9 4 6 2 5 3
n1 n2 n3 n4 n5 n6

Let us consider the following case:

Case 1:

In Insertion sort, you compare the key element with the previous elements. If the previous elements are greater than the key element, then you move the previous element to the next position. Let us make a simple illustration to understand this example:

[9 4 6 2 5 3]

Step 1:

Key =4

The key is compared with the previous element. i.e key is compared with 9.

Since $9 > 4$, move the element 9 to the next position and insert 'key' to the previous position.

Result: [4 9 6 2 5 3]

Step 2:

4	9	6	2	5	3
---	---	---	---	---	---

Key =6; $9 > 6$, move 9 to the next position and insert key to the previous position

Result: [4 6 9 2 5 3]

Step 3:

4	6	9	2	5	3
---	---	---	---	---	---

Key=2

$9 > 2 \rightarrow$ [4 6 2 9 5 3]

$6 > 2 \rightarrow$ [4 2 6 9 5 3]

$4 > 2 \rightarrow$ [2 4 6 9 5 3]

Result : [2 4 6 9 5 3]

Step 4:

2	4	6	9	5	3
---	---	---	---	---	---

Key = 5

$9 > 5 \rightarrow$ [2 4 6 5 9 3]

$6 > 5 \rightarrow$ [2 4 5 6 9 3]

$4 > 5 \neq \rightarrow$ Stop

Result: [2 4 5 6 9 3]

Step 5:

2	4	5	6	9	3
---	---	---	---	---	---

Key=2

$9 > 3 \rightarrow$ [2 4 5 6 3 9]

$6 > 3 \rightarrow$ [2 4 5 3 6 9]

$5 > 3 \rightarrow$ [2 4 3 5 6 9]

$4 > 3 \rightarrow$ [2 3 4 5 6 9]

$2 > 3 \neq \rightarrow$ Stop

Result: [2 3 4 5 6 9]

Now let us consider our insertion sort example. Here, let us discuss the best case analysis first:

In the case of insertion sort, two operations are performed.

1. Scanning through the list, comparing each pair of elements
2. Swaps elements if they are out of order.

As mentioned earlier, best case refers to the minimum time the algorithm takes for 'n' input. The minimum time that the insertion sort can take is when the array is already in the sorted order. Therefore, in the best case, insertion sort runs in $O(n)$ time.

Now let us consider **Worst and Average Case Analysis**:

The worst case for insertion sort will occur when the input list is in decreasing order.

In the case of an insertion sort with input list in the decreasing order, following are the operations performed

1. Scanning through the list, comparing each pair of elements $\rightarrow (1+2+\dots+n-2+n-1)$ scans
2. Swaps elements $\rightarrow (1+2+\dots+n-2+n-1)$ swaps.

i.e it takes $2*(1+2+\dots+n-2+n-1)$ operations to perform insertion sort.

$$\rightarrow 2*(n-1*(n-1+1))/2$$

$$=n*n-1 \rightarrow O(n^2)$$

Therefore, the worst case complexity is $O(n^2)$.

When analyzing algorithms, the average case often has the same complexity as the worst case. So insertion sort, on average, takes $O(n^2)$. We will discuss in detail about estimating the complexity in a short while.

5.1.5 Reasons to analyze algorithm

As with our pizza baking example, we can have multiple ways of preparation of pizza. In the same manner, for a given problem, there may be multiple algorithms. However, in order to determine which algorithm is more efficient than others, we have to analyze the algorithms. This analysis is done by comparing the time required for executing the algorithm (time complexity) or space required for executing the algorithm (space complexity). While considering time complexity, we consider the number of noticeable operations that are carried out by the algorithm.

The amount of time taken for completion of an algorithm is called time complexity of the algorithm. It is a theoretical estimate that measures the growth rate of the algorithm $T(n)$ for a large number of n , the input size. Usually, the time complexity is measured in terms of Big-Oh notation.

5.1.6 Big-Oh notation

Big O notation is a mathematical notation used to describe the upper bound of an algorithm's runtime or space complexity in terms of the input size. It provides an abstract measure of the time or space complexity by focusing on the dominant factors and ignoring constant factors and lower-order terms.

The Big-Oh notation defines that for a large number of input 'n', the growth rate of an algorithm $T(n)$ is of the order of a function 'g' of 'n' as indicated by the following relation:

$$T(n) = O(g(n)). \quad (1)$$

The term 'of the order' means that $T(n)$ is less than a constant multiple of $g(n)$ for $n \geq n_0$. Therefore, for a constant 'c', the relation can be defined as given below:

$$T(n) \leq c \cdot g(n) \text{ where } c > 0 \quad (2)$$

From the above relation, we can say that for a large value of n, the function 'g' provides an upper bound on the growth rate 'T'.

Basic algorithms are usually sequence, selection or iteration. In the case of a sequence, the order of those steps are crucial to ensuring the correctness of an algorithm. In the case of a selection, skipping of some sequences in an algorithm takes place based upon the selection criteria. In the case of iteration, repetition takes place to execute steps a certain number of times or until a certain condition is met. This will become more clear in the following section.

Let us consider the following cases that are used by a programmer to write an algorithm.

1. Simple statement
2. Sequence structure
3. The loop structure
4. If-then-else structure

5.1.6.1 Simple Statement

Let us assume that a statement takes a unit time to execute, i.e., 1. Thus, $T(n) = 1$. The number of steps taken by the above algorithm for completion is one.

$$\text{Thus } T(n) = 1 \quad (3)$$

The above algorithm takes one step to complete.

Now, according to Big Oh notation (O-notation),

$$T(n) = O(g(n)),$$

$$T(n) \leq c \cdot g(n) \text{ where } c > 0. \quad (2)$$

To satisfy the relation 3, the relation 2 can be rewritten as:

$$T(n) \leq 1 \cdot 1$$

where $c = 1$ and $n_0 = 0$. Comparing the above relation with relation 2, we get $g(n) = 1$. Therefore, the above relation can be expressed as:

$$T(n) = O(1)$$

We can say that the time complexity of above algorithm is $O(1)$, i.e., of the order 1 expressed as:

$$T(n) = O(1)$$

Let us take another example where time complexity is more than one.:

Example 1: compute the time complexity of the following relation

$$T(n) = 254$$

Solution

From relation 2,

$$T(n) \leq c \cdot g(n) \text{ where } c > 0$$

From example 1,

$$T(n) = 254$$

$$T(n) \leq 254 \cdot 1$$

$$c = 254, g(n) = 1$$

$$T(n) = O(1)$$

The above example is the case with sequence structure, the details of which are given below.

5.1.6.2 Sequence Structure

Here, the execution time of sequence structure is equal to the sum of execution time of individual statements present within the sequence structure. Consider the following algorithm. It consists of a sequence of statements 1 to 4:

Let us consider the following algorithm to calculate the area of a rectangle.

Algorithm AreaRectangle

{

Step

1. Read length, breadth;
2. Area = length * breadth;
3. Print Area;

4. Stop

}

The number of steps taken by the above algorithm for completion is four.

Thus $T(n) = 4$ (4)

The above algorithm takes four steps to complete.

Now, according to Big Oh notation (O-notation)

$T(n) = O(g(n))$,

$T(n) \leq c \cdot g(n)$ where $c > 0$. (2)

To satisfy the relation 2, the relation can be rewritten as:

$T(n) \leq 4 \cdot 1$ where $c = 4$ and $n_0 = 0$

Comparing the above relation with relation 2, we get $g(n) = 1$. Therefore, the above relation can be expressed as:

$T(n) = O(1)$

We can say that the time complexity of above algorithm is $O(1)$, i.e., of the order 1 expressed as:

$T(n) = O(1)$

5.1.6.3 The loop structure

Let us consider the following algorithm to find the number of ones.

Algorithm Count_ones()

{

Step

1. Num_of_ones = 0;

2. For (I = 1 to N)

{

2.1 Read Num;

2.2 If (Num == 1)

Num_of_ones = Num_of_ones + 1;

}

3. Prompt "The number of ones elements =";

4. Print Num_of_ones;

5. Stop

}

This algorithm takes the following steps for its completion:

No. of simple steps = 4 (Steps 1, 3, 4, 5)

No. of Loops of steps 1 to N = 1

No. of statements within the loop = 3 (1 comparison, 1 addition, 1 assignment)

Thus, $T(n) = 3*N + 4$

Now for $N \geq 4$, we can say that

$$3*N + 4 \leq 3*N + N \leq 4N$$

Therefore, we can say that

$$T(n) \leq 4N \text{ where } c = 4, n_0 \geq 4$$

$$T(n) = O(N)$$

Hence, the given algorithm has time complexity of the order of N, i.e., $O(N)$. This indicates that the term 4 has negligible contribution in the expression: $3*N + 4$.

Note: In case of nested loops, the time complexity depends upon the count of both outer and inner loops. Consider the nested loops given below. This program segment contains 'S', a sequence of statements within nested loops I and J.

For (I = 1; I <= N; I++)

{

For (J = 1; J <= N; J++)

{

S;

}

}

It may be noted that for each iteration of the I loop, the J loop executes N times. Therefore, for N iterations of I loop, the J loop would execute $N*N = N^2$ times. Accordingly, the statement S would also execute N^2 times. Thus,

$$T(n) = N^2$$

To satisfy the relation 2, the above relation can be rewritten as shown below:

$$T(n) \leq 1 * N^2$$

where $c = 1$ and $n_0 > 0$. Comparing the above relation with relation 2, we get $g(n) = N^2$. Therefore, the above relation can be expressed as:

$$T(n) = O(N^2)$$

Hence, the given nested loop has time complexity of the order of N^2 , i.e., $O(N^2)$.

Now, let us calculate the time complexity of the following questions.

Compute the time complexity of the following relations:

(1) $T(n) = 2834$

(2) $T(n) = 9*n + 18$

(3) $T(n) = 18*(N^2) + 7$

(4) $T(n) = 8*(N^3) + 3 N^2 + 6*N$

Solution:

(1) $T(n) = 2834 \leq 2834*1$, where $c = 2834$ and $n_0 = 0$.
 $= O(1)$

Ans. The time complexity = $O(1)$

(2) $T(n) = 9*n + 18 \leq 9*n + n \leq 10 *n$ for $c = 10$ and $n_0 = 18$
 $= O(N)$

Ans. The time complexity = $O(N)$

(3) $T(n) = 18*N^2 + 7 \leq 18*N^2 + N$ for $N = 7$

Now for $N \leq N^2$, we can rewrite the above relation as given below:

$$18*N^2 + N \leq 18*N^2 + N^2 \leq 19 N^2 \text{ for } c = 19 \text{ and } n_0 = 7$$

Thus, $T(n) = O(N^2)$

Ans. The time complexity = $O(N^2)$

(4) $T(n) = 8*(N^3) + 3N^2 + 6*N$

For $N^2 \geq 6*N$, we can rewrite the above relation as given below:

$$8*N^3 + 3N^2 + 6*N \leq 8*N^3 + 3N^2 + N^2 \leq 8*N^3 + 4*N^2$$

For $N^3 \geq 4*N^2$, we can rewrite the above relation as given below:

$$8*N^3 + 4*N^2 \leq 8*N^3 + N^3 \leq 9*N^3 \text{ for } c = 9 \text{ and } n_0 = 6$$

Thus, $T(n) = O(N^3)$

Ans. The time complexity = $O(N^3)$



5.1.6.1 If-then-else structure

In the if-then-else structure, the then and else parts are considered separately. Consider the if-then-else structure given in figure 5.1.1. Let us assume that the 'statement 1' of the part has the time complexity $T1$ and that the 'statement 2' of the else part has the time complexity $T2$. The time complexity of the if-then-else structure is taken to be the maximum of the two, i.e., $\max(T1, T2)$.

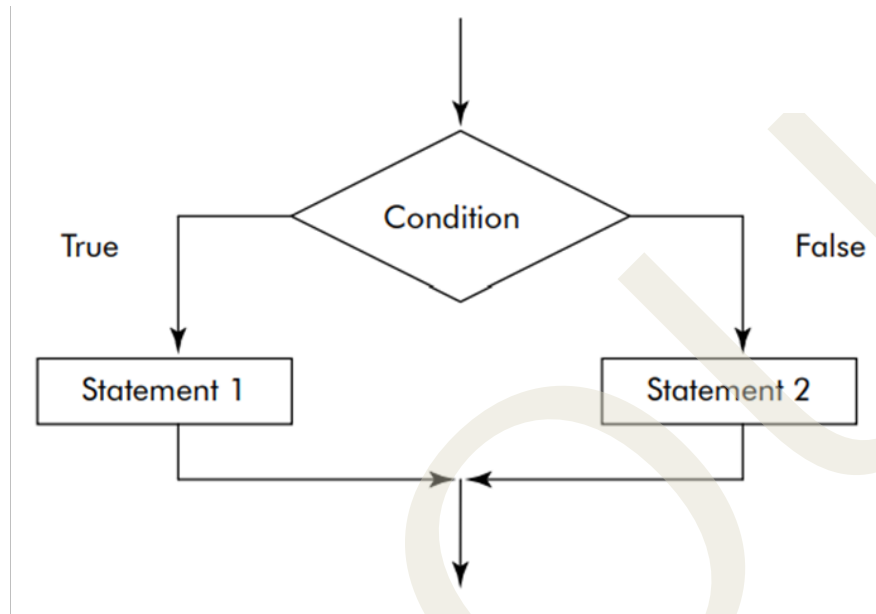


Fig 5.1.1 the if-then-else structure

Example 10: Consider the following algorithm. Compute its time complexity.

```
if (a > b)
{
    a = a - 1;
}
else
{
    for (i=1; i <= N, i++)
    {
        a = a+i;
    }
}
```

Solution: It may be noted that in the above algorithm, the time complexity of 'then' and 'else' parts are $O(1)$ and $O(N)$, respectively. The maximum of the two is $O(N)$.

Therefore, the time complexity of the above algorithm is $O(N)$.

5.1.7 Asymptotic Notations

To communicate complex or large information efficiently and unambiguously, we use signs or symbols to represent ideas, concepts, or quantities in a concise and standardized way. This is called notation. To analyze algorithm performance, we use asymptotic notation.

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. Following are a few of the different types of Asymptotic Notations.

1. Big Oh - O -notation(Asymptotic Upper Bound)
2. Omega - Ω -notation(Asymptotic Lower Bound)
3. Big Theta Θ -notation(Asymptotic Tight Bounds)

While analyzing algorithms, we must consider what happens when the size of the input is large. We usually consider one algorithm to be more efficient if its worst case running time has a lower order of growth. Algorithms are classified into three based on their order of growth. They are:

1. Algorithms that grow at least as fast as some function
2. Algorithms that grow no faster
3. Algorithms that grow at the same rate

The above three categories are usually represented using the Asymptotic Notations Big Omega $\Omega(g(n))$, Big Oh $O(g(n))$, and Big Theta $\Theta(g(n))$ respectively. These notations will be discussed in detail in a short while.

5.1.7.1 Big Oh- O -notation(Asymptotic Upper Bound) - Worst-case

O -notation gives the worst-case complexity of an algorithm. It focuses on how the runtime of an algorithm scales with the size of the input.

Definition

If $f(n)$ is the runtime of your algorithm and $g(n)$ is a time complexity you are trying to relate to your algorithm, then $f(n)$ is $O(g(n))$ if there exist real constants c (where $c > 0$) and n_0 such that: $f(n) \leq c \cdot g(n)$

For all input sizes n (where $n \geq n_0$).

Consider the following functions

$$f(n) = 3n + 2$$

$$g(n) = n$$

We want to show that $f(n) = O(g(n))$.

$$f(n) = 3n + 2$$

Relate $f(n)$ to $g(n)$, $T(n) = 3*n + 2 \leq 3*n + n \leq 4*n$

Determine constants, $c = 4$ and $n_0 = 2$

Thus for $n \geq 2$, $3n+2 \leq 4n$

Worst-case Complexity: Big-O notation helps us understand the worst-case scenario for an algorithm's performance. In this example, as n grows large, the linear term $3n$ dominates the constant term 2 . Therefore, the overall complexity can be simplified to $O(n)$, indicating that the algorithm's runtime increases linearly with the input size.

Constants: The constants c and n_0 are chosen to show that the inequality holds for sufficiently large input sizes. Here, $c=4$ and $n_0=2$ work to demonstrate that $3n+2$ is bounded above by $4n$.

By using Big - Oh notation we can represent the time complexity as follows.

$$3n + 2 = O(n)$$

5.1.7.2 Ω -notation(Asymptotic Lower Bound) - Best case

Ω -notation provides the best case complexity of an algorithm. It focuses on how the runtime of an algorithm behaves in the best possible scenario as the input size grows.

If $f(n)$ is the runtime of your algorithm and $g(n)$ is a time complexity you are trying to relate to your algorithm, then $f(n)$ is $\Omega(g(n))$, if for some real constants c (where $c > 0$) and n_0 (where $n_0 > 0$), $f(n)$ is $\geq c * g(n)$ for every input size n (where $n > n_0$).

Consider the following functions

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq c*g(n)$ for all values of

$$c > 0 \text{ and } n_0 \geq 1$$

$$f(n) \geq c * g(n)$$

$$\Rightarrow 3n + 2 \geq c * n$$

$3n+2 \geq 3n$ Here, $3n$ is a lower bound for $3n+2$ for $n \geq 1$.

Determine constants c and n_0 :

$$c=3$$

$$n_0=1$$

Thus, for $n \geq 1$: $3n+2 \geq 3n$

Above condition is always TRUE for all values of $c = 3$ and $n \geq 1$.

Best-case Complexity: Big-Omega notation helps us understand the best-case scenario for an algorithm's performance. In this example, as n grows large, the linear term $3n$ dominates the constant term 2 . Therefore, the overall complexity can be simplified to $\Omega(n)$, indicating that the algorithm's runtime grows at least linearly with the input size.

Constants: The constants c and n_0 are chosen to show that the inequality holds for sufficiently large input sizes. Here, $c=3$ and $n_0=1$ work to demonstrate that $3n+2$ is bounded below by $3n$.

By using Big - Omega notation we can represent the time complexity as follows

$$3n + 2 = \Omega(n)$$

5.1.7.3 Θ -notation(Asymptotic Tight Bounds) - Average-case

Θ -notation is used for analyzing the average-case complexity of an algorithm. It gives a tight bound on the algorithm's runtime by defining both an upper and lower bound, indicating that the runtime grows asymptotically as a specific function of the input size in both the best and worst cases.

If $f(n)$ is the runtime of your algorithm and $g(n)$ is a time complexity you are trying to relate to your algorithm, then $f(n)$ is $\Theta(g(n))$ if there exist real constants c_1 , c_2 , and n_0 (where $c_1 > 0$, $c_2 > 0$, $n_0 > 0$) such that: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all input sizes n (where $n \geq n_0$).

such that: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all input sizes n (where $n \geq n_0$).

$f(n)$ is $\Theta(g(n))$,

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

Relate $f(n)$ to $g(n)$:

To establish $f(n)$ is $\Theta(g(n))$, we need to find constants c_1 , c_2 , and n_0

such that: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all values of $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$\Rightarrow C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 2$.

If for some real constants c_1 , c_2 and n_0 ($c_1 > 0$, $c_2 > 0$, $n_0 > 0$),

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for every input size n ($n > n_0$).

$f(n)$ is $\Theta(g(n))$ implies $f(n)$ is $O(g(n))$ as well as $f(n)$ is $\Omega(g(n))$.

Lower Bound: $3n+2 \geq 3n$, Here, $c_1=3$.

Upper Bound: $3n+2 \leq 3n+2n \leq 5n$

Average-case Complexity: Big-Theta notation helps us understand the average-case scenario for an algorithm's performance. It tightly bounds the algorithm's runtime, indicating that it grows linearly with the input size, irrespective of minor variations.

Constants: The constants c_1 , c_2 , and n_0 are chosen to show that the inequalities hold for sufficiently large input sizes. Here, $c_1=3$, $c_2=5$, and $n_0=1$ work to demonstrate that $3n+2$ is bounded both above and below by linear functions of n .

By using Big - Theta notation we can represent the time complexity as follows.

$$3n + 2 = \Theta(n)$$

Following table 5.1.1 shows the main difference between the three notations.

Table 5.1.1

Big oh (O)	Big Omega (Ω)	Big Theta (Θ)
Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω takes a small amount of time as compared to Big-O it could possibly take for the algorithm to complete.	Big- Θ take very short amount of time as compare to Big-O and Big- Ω it could possibly take for the algorithm to complete.

Recap

- ◆ Essential Properties of Algorithms
- ◆ Input: Zero or more inputs before execution
- ◆ Output: At least one output after execution
- ◆ Definiteness: Clear, unambiguous steps
- ◆ Finiteness: Terminates after finite steps
- ◆ Effectiveness: Each instruction must be feasible and purposeful
- ◆ Cases During Algorithm Analysis
- ◆ Best Case: Input gives fastest execution
- ◆ Worst Case: Input causes maximum work

- ◆ Average Case: Input leads to average performance
- ◆ Complexity of Algorithms
- ◆ Efficiency measured by:
 - ◆ Time complexity: Time taken to run
 - ◆ Space complexity: Memory used during execution
- ◆ Big-Oh Notation
- ◆ Represents upper bound of growth
- ◆ $T(n) = O(g(n))$, where $T(n) \leq c \cdot g(n)$, for $n \geq n_0$
- ◆ Ignores constants and lower-order terms
- ◆ Basic Algorithm Structures & Their Complexities
- ◆ Simple Statement
- ◆ Constant time $\rightarrow T(n) = O(1)$
- ◆ Sequence Structure
- ◆ Sum of individual steps \rightarrow Still $O(1)$
- ◆ Loop Structure
- ◆ Single loop (e.g., from 1 to N): $O(N)$
- ◆ Nested loop ($N \times N$): $O(N^2)$
- ◆ Big Omega – $\Omega(n) \rightarrow$ Best-Case
- ◆ Represents: Lower bound of runtime
- ◆ Means: Algorithm takes at least this much time
- ◆ Used for: Best-case performance
- ◆ Definition:

$$f(n) \geq c \cdot g(n) \text{ for } n \geq n_0$$
- ◆ Big Theta – $\Theta(n) \rightarrow$ Average-Case
- ◆ Represents: Tight bound of runtime
- ◆ Means: Runtime grows exactly at the same rate
- Used for: Average-case (or exact growth)
- ◆ Definition:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for } n \geq n_0$$

Objective Type Questions

1. What does Big-O notation represent in algorithm analysis?
2. Which asymptotic notation represents the best-case complexity?
3. Big Theta notation provides which type of bound?
4. Write the standard notation for asymptotic upper bound.
5. What is the average-case time complexity of an algorithm represented by?
6. What is the Big-O of $f(n) = 5n + 10$?
7. In Big-O notation, what does the variable n represent?
8. If $f(n) = 3n + 2$, then $f(n)$ is in which asymptotic class?
9. Which notation is used to describe the exact growth of an algorithm?
10. Name the notation used when analyzing the performance for large input size.

Answers to Objective Type Questions

1. Asymptotic upper bound (worst-case)
2. Big Omega (Ω)
3. Tight bound (both upper and lower)
4. $O(g(n))$
5. Big Theta (Θ)
6. $O(n)$
7. Input size
8. $O(n)$, $\Omega(n)$, and $\Theta(n)$
9. Theta (Θ)
10. Asymptotic notation

Assignments

1. Explain the different types of asymptotic notations used in algorithm analysis with suitable examples.
2. Determine the time complexity of the function $f(n) = 4n^2 + 5n + 6$ using Big-O, Big-Ω, and Big-Θ notations.
3. Explain how different loop structures (simple and nested) affect the time complexity of an algorithm.
4. Explain with examples how input size affects the performance of an algorithm and why worst-case analysis is often preferred.

Reference

1. Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
2. Horowitz, E., Sahni, S., & Mehta, D. (2008). *Fundamentals of Data Structures in C++* (2nd ed.). Universities Press.
3. Lafore, R. (2002). *Data Structures and Algorithms in C++* (2nd ed.). Sams Publishing.

Suggested Reading

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). *Data Structures and Algorithms in C++* (2nd ed.). Wiley.
3. Malik, D. S. (2010). *Data Structures Using C++* (2nd ed.). Cengage Learning.

Unit 2

Searching and Sorting

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ to list different types of searching (e.g., linear, binary) and sorting (e.g., bubble, insertion, selection) algorithms used in data structures.
- ◆ to familiarize the step-by-step process of linear search and binary search with examples.
- ◆ to analyze the efficiency of a sorting or searching algorithm for different input sizes
- ◆ to describe the estimation of time complexity for various sorting algorithms
- ◆ to develop a C program that integrates both sorting and searching operations for real-time applications

Prerequisites

Consider a student looking for their hall ticket in a stack of papers just before an exam, or a teacher arranging answer sheets in alphabetical order before entering marks. In both situations, the efficiency of finding or organizing the data can save time and reduce errors. These are simple yet powerful real time examples of searching and sorting in everyday life. Whether it is an ATM machine retrieving your account balance, a music player sorting songs by artist, or a hospital system accessing patient records, the principles of searching and sorting are at work behind the scenes. In computer science, searching means finding the location or presence of a specific data item, while sorting refers to arranging data in a logical sequence, such as from smallest to largest. Algorithms like linear search and binary search help locate items, whereas bubble sort, selection sort, and insertion sort help arrange them. These techniques are the foundation of many applications and are essential for efficient data handling.

Understanding these concepts offers several benefits for learners. It strengthens the ability to think logically and solve problems systematically. By analysing and implementing different algorithms, learners develop a deeper understanding of how computers manage data. These skills are critical for grasping more advanced topics such as database management, memory organization, and algorithm design. Implementing searching and



sorting in a programming language like C not only improves coding skills but also encourages thinking about performance and optimization. Moreover, learners begin to appreciate how algorithmic thinking shapes the functioning of everyday tools and technologies. By mastering searching and sorting, they become better equipped to build efficient software systems and excel in both academic and professional computing environments.

Key words

Linear Search, Binary Search, Bubble Sort, Selection Sort, Insertion Sort, Time complexity, Sequential search, Comparison Based Sorting, Performance Analysis.

Discussion

5.2.1 Searching

Searching is the process of identifying a specific element within a given collection of items. It plays a vital role in both computer science and everyday tasks. For example, locating a book on a shelf organized by title, finding a word in a dictionary sorted alphabetically, or selecting appropriate clothing from a closet all involve different searching strategies. Another common instance is using a search bar in an e-commerce website to find a particular product among thousands of listings. The technique used in each case depends on how the data is arranged and how quickly the result needs to be obtained. An efficient search reduces time and effort, making information access more convenient. A search operation is said to be successful when the target element is located; if not, the result is considered unsuccessful. In computing, the most widely used searching techniques include linear search and binary search.

5.2.2 Linear search

Linear Search, also known as sequential search, is the simplest method of searching in which each element of the list is checked one by one until the desired element is found or the list ends. In a linear search, the algorithm starts from the first element and compares each element with the target value. If a match is found, the position (index) of the element is returned. If the search reaches the end without finding the item, it returns -1 or a message indicating that the item is not present.

Let's say you have the following list of numbers: [10, 5, 8, 2, 15] and you want to find the number 8.

- ◆ **Step 1:** Compare 10 with 8. No match.
- ◆ **Step 2:** Compare 5 with 8. No match.
- ◆ **Step 3:** Compare 8 with 8. Match found! The search stops, and you know 8 is at index 2 (if starting from 0).

5.2.2.1 Algorithm

function linearSearch(array, target):



```

for (i=0; i<=length-1;i++)
    if array[i] == target:
        return i
return -1                // target not found

```

5.2.2.2 Time complexity of linear search

In linear search, we examine each element one by one until the desired element is found or the end of the list is reached. The time complexity of linear search depends on the position of the target element in the array. It is analysed under three cases:

a. Best Case – $O(1)$ - If the element to be searched is at the first position, only one comparison is needed.

Example: Searching for 15 in {15, 20, 32, 45, 23, 50, 40}
Time complexity: $O(1)$

b. Worst Case – $O(n)$ -If the element is at the last position or not present at all, we need to compare all n elements.

Example: Searching for 40 or an element like 60 in the same array.
Time complexity: $O(n)$

c. Average Case – $O(n)$ -In the average case, we assume that the element is equally likely to be found at any position.

- ◆ To search for the 1st element \rightarrow 1 comparison
- ◆ To search for the 2nd element \rightarrow 2 comparisons
- ◆ ...
- ◆ To search for the n th element \rightarrow n comparisons

So, total comparisons = $1 + 2 + 3 + \dots + n = n(n + 1)/2$

Average comparisons =

$$\frac{1 + 2 + 3 + \dots + n}{n} = \frac{n(n + 1)}{2n} = \frac{(n + 1)}{2}$$

Thus, the average case time complexity is:

$$T(n) = O((n + 1)/2) \approx O(n)$$

5.2.2.3 Linear Search: Step-by-Step Working

Let us understand how the **linear search algorithm** works by analyzing the search process for a specific element within an array.

Given Array:

Index: 0 1 2 3 4 5 6

Array: {92, 85, 53, 40, 25, 76, 12}

Suppose we want to search for the element 25. The search begins from the first index (index 0) and proceeds one element at a time until the element is found or the end of the array is reached.

Step 1: Compare with 92

- ◆ Compare 25 with the element at index 0 \rightarrow 92
- ◆ Since $25 \neq 92$, move to the next index.

Step 2: Compare with 85

- ◆ Compare 25 with the element at index 1 \rightarrow 85
- ◆ Since $25 \neq 85$, move to the next index.

Step 3: Compare with 53

- ◆ Compare 25 with the element at index 2 \rightarrow 53
- ◆ Since $25 \neq 53$, move to the next index.

Step 4: Compare with 40

- ◆ Compare 25 with the element at index 3 \rightarrow 40
- ◆ Since $25 \neq 40$, move to the next index.

Step 5: Compare with 25

- ◆ Compare 25 with the element at index 4 \rightarrow 25
- ◆ Since $25 = 25$, the element is found at index 4.
- ◆ The search stops and returns the position.

5.2.3 Binary Search

Have you ever tried to find the meaning of a word using a physical dictionary? Think about how you approach this task. You don't usually start from the very first page and flip through every single word one by one. Instead, you typically open the dictionary somewhere near the middle and look at the words on that page. If your target word comes *before* the words you see, you flip back towards the beginning. If it comes *after*, you flip forward. You keep repeating this process, narrowing down your search to smaller and smaller sections until you land on the exact word you're looking for.

This method reflects the core idea behind Binary Search, one of the most efficient search algorithms used in computer science. Binary Search is an algorithm used to find the position of a specific element within a *sorted* list or array. It drastically reduces the number of comparisons needed to find the desired element by dividing the search space

in half during each step. Because of this, the binary search algorithm is much faster than a linear search, especially when dealing with large datasets.

Example

Suppose you are searching for the number 25 in the following sorted array:

[5, 10, 15, 20, 25, 30, 35, 40]

Initial pointers: low = 0, high = 7

Calculate mid = $(0 + 7) // 2 = 3 \rightarrow$ element at index 3 is 20

$25 > 20 \rightarrow$ search in the right half \rightarrow update low = 4

New mid = $(4 + 7) // 2 = 5 \rightarrow$ element at index 5 is 30

$25 < 30 \rightarrow$ search in the left half \rightarrow update high = 4

New mid = $(4 + 4) // 2 = 4 \rightarrow$ element at index 4 is 25

$25 == 25 \rightarrow$ element found at index 4

The search took only 3 comparisons instead of scanning all 8 elements.

5.2.3.1 Algorithm

function binarySearch(array, target):

 low = 0

 high = length(array) - 1

 while low <= high:

 mid = low + (high - low) / 2

 if array[mid] == target:

 return mid // Target found, return its index

 else if array[mid] < target:

 low = mid + 1 // Target is in the upper half

 else:

 high = mid - 1 // Target is in the lower half

 return -1 // Target not found

5.2.3.2 Step-by-Step Working of Binary Search

Let's break down how the binary search algorithm works, step by step:

1. Start with a sorted list.

Binary search only works on data that is already sorted in increasing or decreasing order. If the data is unsorted, it must be sorted first.

Set two pointers: low and high.

- ◆ low points to the beginning of the list (index 0).
- ◆ high points to the end of the list (last index).

2. Find the middle element.

- ◆ Calculate the middle index: $\text{mid} = (\text{low} + \text{high}) // 2$
- ◆ Compare the element at index mid with the target value.

3. Three possible outcomes:

- ◆ If the middle element **matches** the target, the search is successful. Return the index.
- ◆ If the target is **less than** the middle element, update $\text{high} = \text{mid} - 1$ to search in the **left half**.
- ◆ If the target is **greater than** the middle element, update $\text{low} = \text{mid} + 1$ to search in the **right half**.

4. Repeat the process until:

- ◆ The target is found, or
- ◆ low becomes greater than high, which means the element is not in the list.

5.2.3.3 Time complexity of Binary Search

The time complexity of the binary search algorithm can be analyzed in terms of best case, worst case, and average case scenarios.

a. Best Case Time Complexity

The target element is found at the middle of the array on the first comparison. In the best case, binary search finds the target element in the first comparison itself, requiring only one operation. So time complexity is $O(1)$.

b. Worst Case Time Complexity

The target element is not in the array or is located at the very end of the search process. In the worst case, the search space is halved each time, leading to the maximum number of comparisons being equal to the logarithm (base 2) of the number of elements in the array. Thus, the time complexity is $O(\log_2 n)$.

c. Average Case Time Complexity

The target element is found at some point during the search, but not necessarily at the first comparison. On average, the binary search will need to reduce the search space logarithmically. Hence, the average time complexity is also $O(\log_2 n)$.

5.2.4 Sorting

Observe the image below. The boys are carefully arranging books on a shelf. This act of placing the books in a particular order is an example of sorting in real life. The criteria they use for sorting could vary. They might arrange the books alphabetically by the author's name, by subject such as Science, Arts, or History, by size, or even by color or publication date. No matter the method, the main goal is to organize the books in a systematic and meaningful way that makes them easier to find and use.



Fig 5.2.1 Sorting books

In a similar way, sorting plays an important role in computers and digital systems. In the modern digital age, we often deal with large sets of data, such as lists of student records, customer details, or product information. Suppose we have a list of student data and we want to arrange the students based on their age. To do this quickly and accurately, we use specific methods known as sorting algorithms.

A sorting algorithm is a clear and structured method used to arrange data in a specific order, usually either in increasing or decreasing order, based on a chosen characteristic or value. Sorting improves the efficiency of data handling, searching, and presentation. When data is sorted, it becomes easier to analyze, interpret, and manage.

Computer science provides us with many types of sorting techniques, each designed for different types of data and use cases. In this section, we will learn about three basic and widely used sorting methods called selection sort, bubble sort and insertion sort. These methods are simple and easy to understand, and they lay the foundation for more advanced sorting strategies used in real world applications.

5.2.5 Selection Sort

Imagine a teacher who is organizing students for a class photograph. To make it visually appealing, the teacher decides to arrange the children in order of height, from the shortest to the tallest. The teacher begins by identifying the shortest child in the group and asks them to stand in the first position. Next, the teacher finds the next shortest child among the remaining students and places them in the second position. This process continues until every child is arranged in the correct order. This real-life strategy is very

similar to how the Selection Sort algorithm works. It is simple, systematic, and easy to understand.

Selection sort is a comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the order) element from the unsorted portion of the list and placing it at the beginning of the sorted portion. The algorithm gets its name because it repeatedly **selects** the smallest element from the unsorted segment.

The logic of Selection sort for sorting a list of integers is given below.

Step1. First find the smallest element in the given list.

Step2. Swaps it with the first element of the unordered list.

Step3. Find the second smallest element.

Step4. Swaps it with the second element of the unordered list.

Similarly, it continues to sort the given elements.

5.2.5.1 Algorithm

index is a variable to store the index of minimum element

j is a variable to traverse the unsorted sub-array

temp is a temporary variable used for swapping

for (i = 0 ; i < n-1 ; i++)

{

 index = i;

 for(j = i+1 ; j < n ; j++)

 {

 if (A[j] < A[index])

 index = j;

 }

 temp = A[i];

 A[i] = A[index];

 A[index] = temp;

}

5.2.5.2 Step-by-Step logic of Selection Sort

Step-by-Step Logic of Selection Sort

Let's assume we are sorting a list of numbers in **ascending order**:

Given List:

[29, 10, 14, 37, 13]

The algorithm follows these steps:

1. Step 1: Find the smallest element in the entire list.
 - The smallest is 10.
 - Swap it with the first element 29.
 - List becomes: [10, 29, 14, 37, 13]
2. Step 2: Find the smallest element in the remaining list starting from the second position.
 - The smallest is 13.
 - Swap it with 29.
 - List becomes: [10, 13, 14, 37, 29]
3. Step 3: Repeat the process for the remaining unsorted part.
 - The smallest is 14, already in place.
 - List remains: [10, 13, 14, 37, 29]
4. Step 4: Next smallest is 29, found in the last two elements.
 - Swap with 37.
 - List becomes: [10, 13, 14, 29, 37]
5. Now the list is completely sorted.

5.2.5.3 Time Complexity of Selection Sort

1. Best Case Time Complexity: $O(n^2)$

Even if the array is already sorted, selection sort still checks all remaining elements in each pass to find the minimum, leading to the same number of comparisons.

2. Average Case Time Complexity: $O(n^2)$

For randomly ordered elements, the algorithm still performs nested loops with approximately $\frac{n(n-1)}{2} \times 2 = n(n-1)$ comparisons.

3. Worst Case Time Complexity: $O(n^2)$

Even in reverse order, the algorithm still makes the same number of comparisons and swaps.

5.2.6 Bubble sort

Imagine a group of children standing in a line, and the teacher wants to arrange them in order of height, from the shortest to the tallest. The teacher looks at the first two children, and if the one on the left is taller than the one on the right, they are asked to

switch places. Then the teacher moves one step to the right and compares the next pair, continuing this process all the way to the end of the line. At the end of one full pass, the tallest child “bubbles up” to the last position. The teacher repeats this process for the remaining children, ignoring the last child each time, since they are already in the correct place.

This real-life approach is similar to the working of the **Bubble Sort algorithm**, where adjacent elements are compared and swapped if they are in the wrong order. This is repeated until the entire list is sorted. Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly stepping through the list, comparing adjacent pairs of elements, and swapping them if they are in the wrong order. The process is repeated until no more swaps are needed, which means the list is sorted. Bubble sort gets its name because during each pass, the largest (or smallest, depending on order) value “bubbles” to the end of the list.

5.2.6.1 Algorithm

// temp is used to swap elements

```
for (i = 0; i < n - 1; i++) {  
    for (j = 0; j < n - i - 1; j++) {  
        if (A[j] > A[j + 1]) {  
            temp = A[j];  
            A[j] = A[j + 1];  
            A[j + 1] = temp;  
        }  
    }  
}
```

5.2.6.2 Step-by-Step Logic of Bubble Sort

Let's take a list: [29, 10, 14, 37, 13]

Pass 1

- ◆ Compare 29 and 10 → swap → [10, 29, 14, 37, 13]
- ◆ Compare 29 and 14 → swap → [10, 14, 29, 37, 13]
- ◆ Compare 29 and 37 → no swap
- ◆ Compare 37 and 13 → swap → [10, 14, 29, 13, 37]

Largest element (37) is now at the correct position

Pass 2

- ◆ Compare 10 and 14 → no swap



- ◆ Compare 14 and 29 → no swap
- ◆ Compare 29 and 13 → swap → [10, 14, 13, 29, 37]

Second largest element (29) is now in place

Pass 3

- ◆ Compare 10 and 14 → no swap
- ◆ Compare 14 and 13 → swap → [10, 13, 14, 29, 37]

Pass 4

- ◆ Compare 10 and 13 → no swap

List is now fully sorted

5.2.6.3 Time Complexity of Bubble Sort

Table 5.2.1 Complexity of bubble sort

Case	Time Complexity	Explanation
Best Case	$O(n)$	List is already sorted; only one pass needed (with optimized version)
Average Case	$O(n^2)$	Nested loops perform all comparisons
Worst Case	$O(n^2)$	List is in reverse order; maximum number of swaps and comparisons

5.2.7 Insertion sort

Imagine you are arranging playing cards in your hand. You pick one card at a time and place it in the correct position among the already sorted cards in your hand. Initially, you consider the first card as sorted. Then, you take the second card and place it to the left or right depending on its value. You continue this process for all the remaining cards until the entire set is sorted.

This real-life example perfectly illustrates the logic of the Insertion Sort algorithm. It builds the final sorted array one item at a time by comparing each new element with the already sorted portion and inserting it into its correct position. Insertion Sort is a simple, comparison-based sorting algorithm that works similarly to how we sort cards in our hand. It divides the list into a sorted and an unsorted part. Elements from the unsorted part are picked one by one and inserted into their correct position in the sorted part.

Step1: Iterate from **A[0]** to **A[n-1]** over the array.

Step 2: Compare the current element (key) to its predecessor.

Step 3: If the key element is smaller than its predecessor, compare it to the elements before.

Move the greater elements one position up to make space for the swapped element.

5.2.7.1 Algorithm

#A is the given array

variable to traverse the array A

#key = variable to store the new number to be inserted into the sorted sub-array

j = variable to traverse the sorted sub-array

for (i = 1 ; i < n ; i++)

```
{  
    key = A [ i ];  
    j = i - 1;  
    while(j > 0 && A [ j ] > key)  
    {  
        A [ j+1 ] = A [ j ];  
        j--;  
    }  
    A [ j+1 ] = key;  
}
```

5.2.7.2 Step-by-Step logic of Insertion sort

Let's sort this list:

[29, 10, 14, 37, 13]

Step 1:

- ◆ First element 29 is considered sorted.
- ◆ Key = 10
 - Compare with 29 → $29 > 10$
 - Shift 29 to the right
 - Insert 10 at index 0

List becomes: [10, 29, 14, 37, 13]

Step 2:

- ◆ Key = 14

- Compare with 29 → $29 > 14$ → shift 29
- Compare with 10 → $10 < 14$ → insert 14 after 10

List becomes: [10, 14, 29, 37, 13]

Step 3:

- ◆ Key = 37
- Compare with 29 → $29 < 37$ → insert 37 in same place

List remains: [10, 14, 29, 37, 13]

Step 4:

- ◆ Key = 13
- Compare with 37, 29, 14 → all > 13 → shift them
- Insert 13 after 10

List becomes: [10, 13, 14, 29, 37]

5.2.7.3 Time complexity of Insertion sort

Table 5.2.2 Complexity of insertion sort

Case	Time Complexity	Explanation
Best Case	$O(n)$	Already sorted list; only one comparison per element
Average Case	$O(n^2)$	Elements in random order; may shift many elements for each insertion
Worst Case	$O(n^2)$	Reverse order; every element needs to be compared and shifted

Recap

- ◆ Searching is a process used to find a specific item from a group of items stored in a data structure like an array or list.
- ◆ The effectiveness of searching depends on how the data is arranged and how fast the result is needed.
- ◆ Linear search is a basic search method that checks each element one by one from the beginning until the target element is found or the list ends.
- ◆ The number of comparisons in linear search depends on the position of the target; if it is at the beginning, it takes less time, and if it is at the end or not present, it takes more time.
- ◆ Binary search is a faster technique that works only when the list is sorted in ascending or descending order.

- ◆ In binary search, the search range is divided into two halves in each step, and the middle element is compared with the target.
- ◆ Sorting is the process of arranging data in a specific order, such as increasing or decreasing, to improve organization and efficiency.
- ◆ Sorting makes it easier to analyze, search, and manage data efficiently.
- ◆ Selection sort works by selecting the smallest element from the unsorted portion and placing it in the correct position, repeating this for each position in the array.
- ◆ The total number of comparisons in selection sort is the same, regardless of the original order of the data.
- ◆ The time complexity of selection sort is $O(n^2)$ in the best, average, and worst cases, because it always uses nested loops.
- ◆ Bubble sort compares adjacent elements and swaps them if they are in the wrong order, repeating this process multiple times until no more swaps are needed.
- ◆ During each pass in bubble sort, the largest unsorted element moves to its correct position at the end of the list.
- ◆ Insertion sort builds the final sorted list one element at a time by inserting each element into its correct position in the already sorted part of the array.
- ◆ In each step, insertion sort compares the new element with the sorted portion and shifts larger elements one position right to make space.
- ◆ The time complexity of insertion sort in the best case is $O(n)$, when the list is already sorted and only one comparison is needed per element.
- ◆ The average and worst-case complexities are $O(n^2)$, especially when the list is in reverse order and many shifts are required for each insertion.
- ◆ Sorting is essential in computing because it enhances the performance of search operations and ensures faster data access and processing.
- ◆ Understanding the basic searching and sorting techniques is fundamental for solving more complex problems in computer science and programming.

Objective Type Questions

1. Name the search algorithm that checks each element from beginning to end.
2. State the type of list required for binary search to work correctly.

3. Identify the sorting algorithm where the smallest element is repeatedly selected and placed at the correct position.
4. Which search method divides the list into halves?
5. What is the best-case time complexity of linear search?
6. What is the average case time complexity of binary search?
7. Name the sorting technique where the largest element moves to the end after each pass.
8. What is the best-case time complexity of bubble sort (optimized)?
9. Which sorting algorithm is most suitable for small or nearly sorted datasets?
10. What is the return value of binary search if the element is not found?
11. Which sorting algorithm performs well when the input list is already sorted?
12. Point out the sorting algorithm that is inefficient for large datasets due to nested loops.
13. **State** the method used to find the correct position of an element during insertion sort.
14. **Write** the number of comparisons made in the best case for insertion sort.

Answers to Objective Type Questions

1. Linear
2. Sorted
3. Selection
4. Binary
5. $O(1)$
6. $O(\log n)$
7. Bubble
8. $O(n)$
9. Insertion

10. -1
11. Insertion
12. Bubble
13. Shifting
14. n-1

Assignments

1. Explain the working of linear search and binary search algorithms with suitable examples. Compare their performance in terms of time complexity and data requirements.
2. Write an algorithm for binary search. Explain how the middle element is calculated, and how the search space is updated based on comparisons. Provide an example with a sorted list.
3. Describe the process of selection sort with a step-by-step example.
4. What is bubble sort? Explain its working principle with an example. Analyze the efficiency of the algorithm and suggest scenarios where it is not suitable.
5. Discuss the insertion sort algorithm in detail. Explain how it works, its best and worst case scenarios, and why it is efficient for nearly sorted data.

Reference

1. Padma Reddy, G.A. V. (2017). *Data Structures Using C*. Scitech Publications.
2. Thareja, R. (2014). *Data Structures Using C* (2nd ed.). Oxford University Press..
3. Tenenbaum, A. M., Langsam, Y., & Augenstein, M. J. (2006). *Data Structures Using C* (1st ed.). Pearson Education.
4. Kanetkar, Y. (2018). *Data Structures Through C* (2nd ed.). BPB Publications.

Suggested Reading

1. Jain, D. P., & Sharma, M. (2022). *Data Structures Through C: A Practical Approach*. Vikas Publishing.
2. Joshi, R. (2021). *Mastering Data Structures in C*. BPB Publications.
3. Malik, D. S. (2020). *Data Structures Using C* (2nd ed.). Cengage Learning.
4. Bhatt, C., & Dave, M. (2021). An analysis of time complexity of basic sorting algorithms with hybrid approaches. *International Journal of Computer Applications*, 183(1), 1–7.

SGOU

Unit 3

Divide and Conquer Algorithms: Minmax, Binary Search, Quicksort

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ describe how to design a divide and conquer algorithms
- ◆ introduce quicksort
- ◆ familiarize binary search
- ◆ make aware of minimax algorithm
- ◆ familiarize Prim's algorithm and Kruskal's algorithm

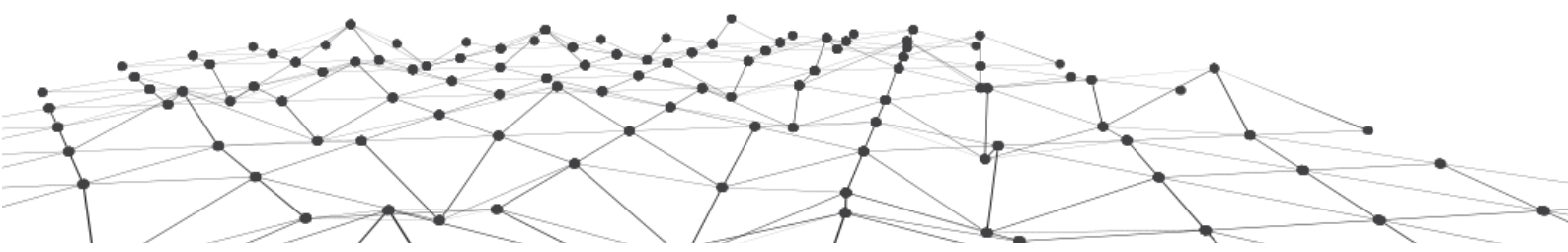
Prerequisites

The prerequisite for learning algorithms is that the student must be aware of at least one computer programming language, and he/she should understand different operations in the data structure. When a student starts learning the divide and conquer, min-max, binary search, and quick sort algorithms, he/she should be aware of the knowledge of trigonometry and spatial geometry.

Learning to divide and conquer algorithms, he/she must know the searching and sorting techniques in algorithms and understand the complexity in the development part of an algorithm. To learn the mini-max algorithm, you have to know the backtracking process in the algorithm design and the construction of the depth-first traversal and free-search algorithm. He/she should know the fundamental concept of non-linear data structures like trees and graphs, so students will quickly understand the binary search and quick sort algorithms.

Key Concepts

Divide and conquer algorithm, Design of divide and conquer algorithm, Minmax algorithm, Binary search, Quick sort



Discussion

5.3.1 Divide and Conquer Algorithm

The divide and conquer is an algorithmic pattern. In algorithmic methods, the design takes a dispute on a considerable input, breaks the information into minor pieces, decides the problem on each small amount, and then merges the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer strategy.

The divide and conquer technique has the following strategy. The method divides the problem instance into two or more minor examples of the same problem, solves the more minor models recursively, and assembles the solutions to explain the original illustration. The recursion stops when an instance is reached which is too small to divide. When dividing the model, one can either use whatever division comes most readily to hand or invest time in dividing carefully to simplify the assembly.

Generally, divide-and-conquer algorithms have three parts.

1. **Divide** : Break the problem into several sub-problems that are more minor instances of the same problem.
2. **Conquer** : Solve each of the subproblems recursively. If a subproblem is small enough or trivial, solve it directly.
3. **Combine** : Combine the solutions of the subproblems to form a solution to the original problem

Figure 5.3.1 is the graphical representation of the divide and conquers algorithm. This technique is used to design algorithms that solve problems so that the problem is broken down into one or more minor instances of the same problem, and each smaller model is solved recursively. These more minor instances of the bigger problem are called sub-problems. The method keeps on breaking the sub-problems to a level at which one can solve them directly. Finally, all the solutions to these sub-problems are combined to get the answer for the main problem. Depending upon the size of the sub-problem, there are two cases: recursive case and base case. The sub-problem size is large enough in the recursive case, and they have to be solved recursively. However, in the base case, the sub-problem size is small enough to be solved directly. Many computer algorithms are based on the divide and conquer approach. They are, maximum and minimum problem, binary search, sorting (merge sort, quick sort), and tower of Hanoi.

There are two fundamental concepts in the divide and conquer strategy, they are relational formula and stopping condition. The relational formula is the formula that generates from the given technique. After the generation of the formula applying the divide and conquer strategy, i.e. break the problem recursively and solve the broken sub-problems. When breaking the problem using the divide and conquer strategy, the method needs to know how much time to apply the divide and conquer technique. The condition where the process needs to stop the recursion steps of the divide and conquer strategy is called a stopping condition.

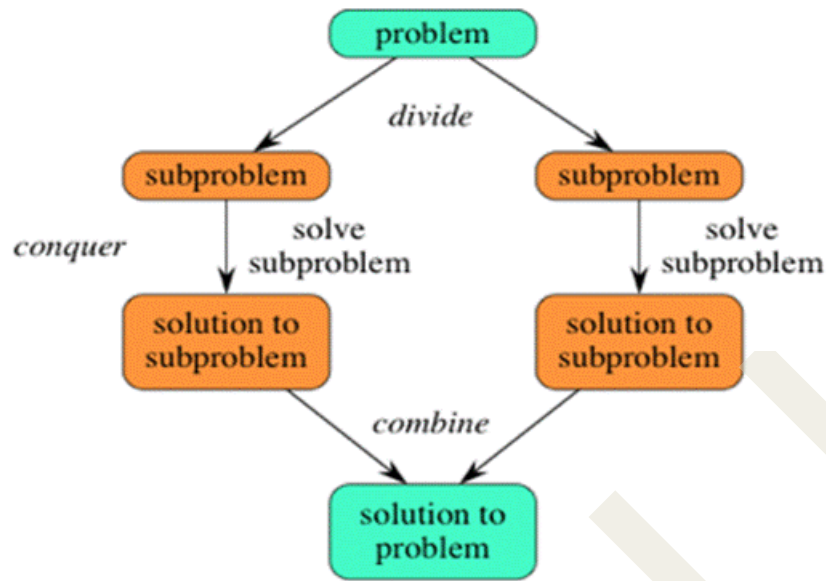


Fig 5.3.1 Divide and Conquer

5.3.1.1 Advantages of Divide and Conquer Algorithm

- ◆ The divide and conquer paradigm allows us to solve complex and often impossible-looking problems, such as the Tower of Hanoi, a mathematical game or puzzle.
- ◆ The divide and conquer method reduces the degree of difficulty since it divides the problem into subproblems that are easily solvable and usually run faster than other algorithms.
- ◆ The divide and conquer algorithm often plays a part in finding other efficient algorithms, and in fact, it was the central role in finding the quick sort and merge sort algorithms.
- ◆ The divide and conquer method uses memory caches effectively. This is because when the sub-problems become simple enough, they can be solved within a cache without having to access the slower main memory, which saves time and makes the algorithm more efficient.
- ◆ The divide and conquer method can even produce more precise computations with rounded arithmetic than iterative methods would.

5.3.1.2 Disadvantages of Divide and Conquer Algorithm

- ◆ The divide and conquer algorithm issue is that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
- ◆ The divide and conquer method sometimes can become more complicated than a basic iterative approach, especially in large sub-problems.
- ◆ In the divide and conquer method sometimes once the problem is broken down into sub-problems, the same sub-problem can occur many times.

- ◆ In the divide and conquer method it can often be easier to identify and save the solution to the repeated sub-problem, which is commonly referred to as memorization.
- ◆ The divide and conquer algorithm may require more memory space due to recursion

5.3.1.3 Basic Method in Divide and Conquer Algorithm

In the divide and conquer approach, the problem is divided into smaller sub-problems, and then each problem is solved independently. Dividing the sub-problems into even smaller sub-problems, finally, the process reaches a stage where no more division is possible. The solution of all the sub-problems is finally merged to obtain the solution of an original problem. A divide and conquer approach has three basic steps. They are, divide/break, conquer/solve, and merge/combine. Figure 5.3.2 represents the three basic fundamental steps in the divide and conquer algorithm.

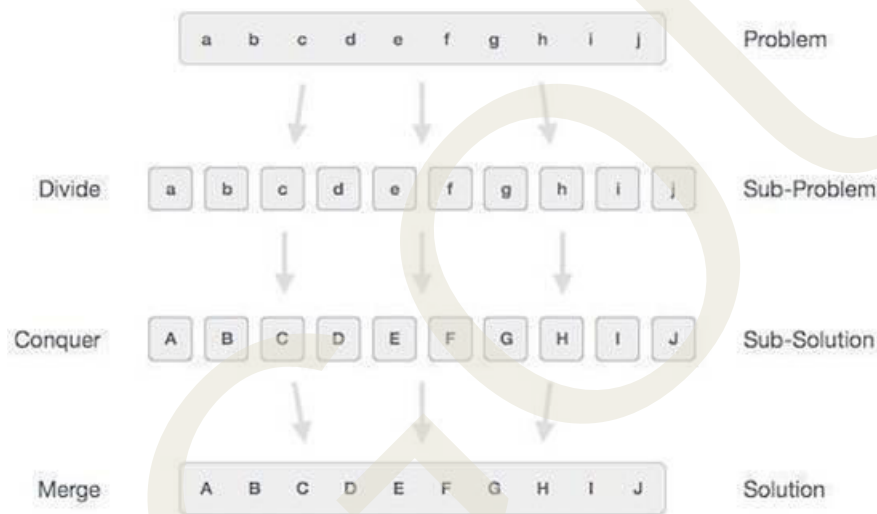


Fig 5.3.2 Three Basic Steps in Divide and Conquer Algorithm.

Divide/Break: This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic but still represent some part of the actual problem.

Conquer/Solve: This step receives many smaller subproblems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine: When the smaller subproblems are solved, this stage recursively combines them until they formulate a solution to the original problem. This algorithmic approach works recursively and merges steps so close that they appear as one.

5.3.2 Binary Search Algorithm

The binary search algorithm is the search technique, which works efficiently on sorted

lists. However, the searching process works only on a sorted set of elements. The binary search algorithm follows the divide and conquer approach in which the sorting list is divided into two halves, and the item is compared with the middle element of the list. If the matching item is found in the list then, the location of the central element is returned; otherwise, search into either of the half depending upon the result produced through the match.

When a binary search algorithm is used to perform operations on a sorted set, the number of iterations can permanently be reduced based on the value that is being searched.

5.3.2.1 Binary Search Algorithm Implementation Steps

The binary search cannot be used for a list of elements arranged in a random order. This searching process is applied only to a sorted list of elements. The binary search process starts comparing the search element with the middle element in the list. The binary search algorithm is implemented using the following steps.

Step 1: Read the search element from the user.

Step 2: Find the middle element in the sorted list.

Step 3: Compare the search element with the middle element in the sorted list.

Step 4: If both are matched, then display “Given element is found” and terminate the function.

Step 5: If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6: If the search element is smaller than the middle element, repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.

Step 7: If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until sub list contains only one element.

Step 9 - If that element also doesn't match with the search element, then display “Element is not found in the list” and terminate the function.

Example 1

Consider the following list of elements and the element to be searched.



Fig 5.3.3 Array of integers

Step1: search element (12) is compared with middle element (50)

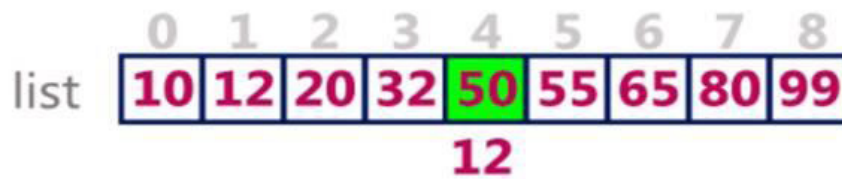


Fig 5.3.4 Array of integers

Both are not matching and element 12 is smaller than 50. So search only in the left sub list (that is 10, 12, 20 and 32).



Fig 5.3.5 Array of integers

Step2: search element (12) is compared with the middle element (12)



Fig 5.3.6 Array of integers

Both are matching. Therefore, the result is “Element found at index 1”

Example 2

Consider the following list of elements and the element to be searched.

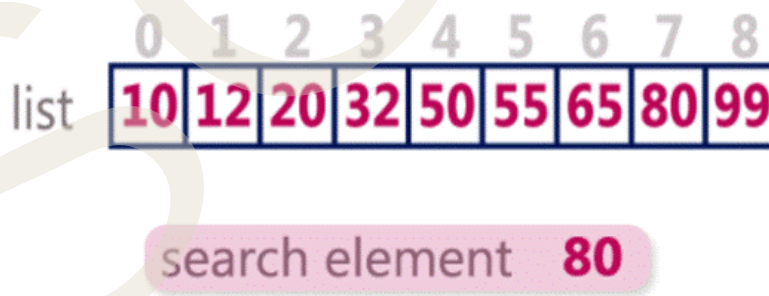


Fig 5.2.7 Array of integers

Step1: search element (80) is compared with the middle element (50)

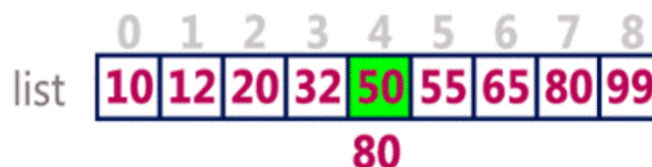


Fig 5.3.8 Array of integers

Both are not matching. The element (80) is larger than element (50). So, search only in the right sub-list (that is, 55, 65, 80 and 99).



Fig 5.3.9 Array of integers

Step 2: search element (80) is compared with the middle element (65)



Fig 5.3.10 Array of integers

Both are not matching. The element 80 is larger than 65. So search only in the right sub list (that is 80 and 90).



Fig 5.3.11 Array of integers

Step 3: search element (80) is compared with the middle element (80).



Fig 5.3.12 Array of integers

Both are matching. So the result is “Element is found at index 7”.

The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

5.3.3 Merge Sort

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting

the lowest values first, resulting in a sorted array.

The breaking down and building up of the array to sort the array is done recursively.

In the animation above, each time the bars are pushed down represents a recursive call, splitting the array into smaller pieces. When the bars are lifted up, it means that two sub-arrays have been merged together.

The Merge Sort algorithm can be described like this:

1. Divide the array into two halves.
2. Recursively apply Merge Sort to each half.
3. Merge the two sorted halves into a single sorted array.
4. Repeat until the entire array is sorted.

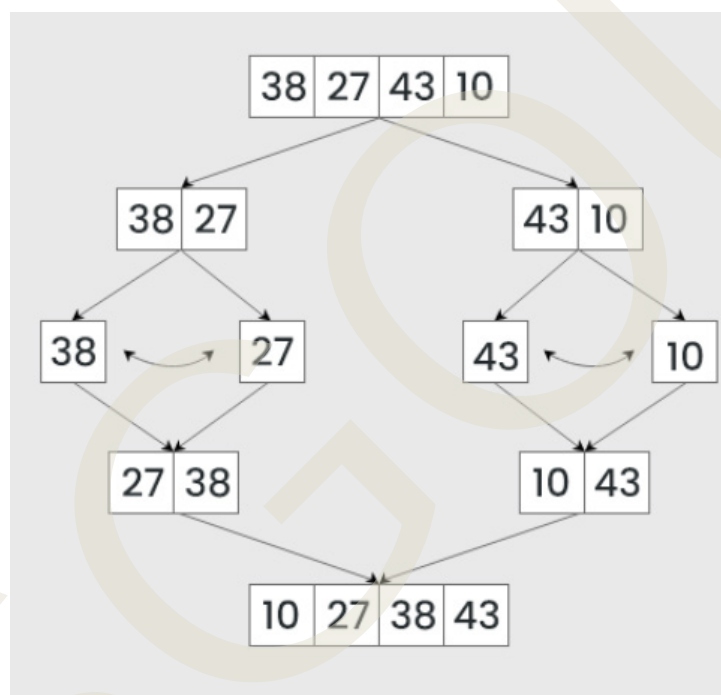


Fig 5.3.13 Merge sort

Example for merge sort

Given array:

[8, 4, 7, 3, 10, 2]

Step 1:

Split into [8, 4, 7] and [3, 10, 2]

Step 2:

Split [8, 4, 7] into [8] and [4, 7]

[4, 7] → [4], [7]

Merge [4] and [7] \rightarrow [4, 7]

Merge with [8] \rightarrow [4, 7, 8]

Split [3, 10, 2] into [3] and [10, 2]

[10, 2] \rightarrow [10], [2]

Merge [10] and [2] \rightarrow [2, 10]

Merge with [3] \rightarrow [2, 3, 10]

Final Merge:

[4, 7, 8] and [2, 3, 10] \rightarrow [2, 3, 4, 7, 8, 10]

Time Complexity

- ◆ Best Case: $O(n \log n)$
- ◆ Average Case: $O(n \log n)$
- ◆ Worst Case: $O(n \log n)$
- ◆ Space Complexity: $O(n)$ (requires temporary arrays)

Advantages

- ◆ Stable sort (preserves order of equal elements)
- ◆ Consistent performance ($O(n \log n)$)
- ◆ Great for large datasets and linked lists

Disadvantages

- ◆ Requires extra memory
- ◆ Slower than in-place sorting algorithms for small arrays

5.3.4 Quick Sort Algorithm

British computer scientist Tony Hoare developed the algorithm in 1959. The name “Quicksort” comes because quick sort can sort a list of data elements significantly faster (twice or thrice faster) than any of the standard sorting algorithms. It is one of the most efficient sorting algorithms and is based on the splitting of an array (partition) into smaller ones and swapping (exchange) based on the comparison with the “pivot” element selected. Due to this, quick sort is also called a “Partition Exchange” sort.

The quicksort is a divide and conquer algorithm. Like all the divide and conquer algorithms, it first divides a larger array into smaller sub-arrays and recursively sorts the sub-arrays. Three steps are involved in the whole sorting process of the quick sort algorithm they are;

1. Pivot selection: Pick an element from the array (usually the leftmost or the rightmost elements of the portion).

2. Partitioning: Reorder the array such that all elements with a value less than the pivot come before the pivot. In contrast, all elements with values greater than the pivot come after it. The equal values can go either way after this partitioning. The pivot is in its final position.
3. Recur: Recursively apply the above steps to the sub-array of elements with smaller values in the pivot and separately to the sub-array of elements with higher values than the pivot.

5.3.4.1 Quick Sort Algorithm Implementation Steps

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it uses divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. The pivot element is one of the elements in the list. The list is divided into two partitions such that “all elements to the left of the pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot”.

Step 1: Consider the first element of the list as pivot

(i.e., Element at first position in the list).

Step 2: Define two variables ‘i’ and ‘j’.

Set ‘i’ and ‘j’ to first and last elements of the list respectively.

Step 3: Increment ‘i’ until $\text{list}[i] > \text{pivot}$ then stop.

Step 4: Decrement ‘j’ until $\text{list}[j] < \text{pivot}$ then stop.

Step 5: If ‘i’ < ‘j’ then exchange $\text{list}[i]$ and $\text{list}[j]$.

Step 6: Repeat steps 3, 4 and 5 until ‘i’ > ‘j’.

Step 7: Exchange the pivot element with $\text{list}[j]$ element.

Example 1

Consider the following unsorted list of elements.



Fig 5.3.14 Array of integers

Define pivot, left and right. Set $\text{pivot} = 0$, $\text{left} = 1$ and $\text{right} = 7$. Here ‘7’ indicates ‘size-1’



Fig 5.3.15 Array of integers

Compare the list[left] with the list[pivot]. if the list[left] is greater than the list[right] then stop the left otherwise move left to the next. Compare the list[right] with the list[pivot]. If the list[right] is smaller than the list[pivot] then stop right otherwise move right to the previous. Repeat the same until left>=right.

If both left and right are stopped but left<right then swap the list[left] with the list[right] and continue the process. If left>=right then swap the list[pivot] with the list[right].



Fig 5.3.16 Array of integers

Compare the list[left]< the list[pivot] as it is true increment left by one and repeat the same, left will stop at 8. Compare the list[right]>the list[pivot] as it is true, decrement right by one and repeat the same, right will stop at 2.



Fig 5.3.17 Array of integers

Here left and right both are stopped and the left is not greater than right so swap the list [left] and list [right].



Fig 5.3.18 Array of integers

Compare the list[left]<list[pivot] as it is true increment left by one and repeat the same, left will stop at 6. Compare the list[right]>list[pivot] as it is true, decrement right by one and repeat the same, right will stop at 4.



Fig 5.3.19 Array of integers

Here left and right both are stopped and left is greater than right so we need to swap the list[pivot] and list[right].



Fig 5.3.20 Array of integers

Here all the numbers to the left side of 5 are smaller and the right side are greater. That means 5 is placed in its correct position.

Repeat the same process on the left sub list and right sub list to the number 5



Fig 5.3.21 Array of integers

In the left sub list as there are no smaller numbers than the pivot left will keep on moving to the next and stop at the last number. AS the list[right] is smaller, right stops at the same position. Now left and right both are equal so swap pivot with right.



Fig 5.3.22 Array of integers

In the right sub list left is greater than the pivot, left will stop at the same position. As the list[right] is greater than the list[pivot], right moves towards left and stops at pivot number position. Now left>right so swap the pivot with right. (the 6 is swapped by itself).



Fig 5.3.23 Array of integers

Repeat the same recursively on both the left and right sub list until all the numbers are sorted. The final sorted list will be as follows.



Fig 5.3.24 Array of integers

The quick sort provides a fast and methodical approach to sorting any lists of things. Following are some of the applications where quick sort is used. Many government and private organizations use the quick sort to arrange accounts or profiles by name or any given ID and sort transactions by time or location. Complex numerical computations and the fastest information search can be done with the help of quick sort.

5.3.5 Min-max Algorithm

Mini-max is a decision-making algorithm typically used in a turn-based, two-player game. The goal of the algorithm is to find the optimal next move. In the algorithm, one player is called the maximizer, and the other player is a minimizer. Both these players

play the game as one tries to get the highest score possible or the maximum benefit while the opponent tries to get the lowest score or the minimum help. Every game has an evaluation score assigned to it to select the maximized value, and the minimiser will determine the minimized value with counter moves. If the maximiser has the upper hand, then the board score will be a positive value, and if the minimiser has the upper hand, then the board score will be a negative value. Therefore, for one player to win, the other has to lose.

Definition: The Mini-max algorithm is a recursive or backtracking algorithm used in decision-making and game theory. It provides an optimal move for the player, assuming that the opponent is also playing optimally.

Figure 5.3.25 is an example of the min-max algorithm; from this figure, there are two players: the maximizer and the other is called minimizer. The maximizer will try to get the maximum possible score, and the minimizer will try to get the minimum possible score. Then, at the terminal node, the terminal values are given, finally, compare those values and backtrack the tree until the initial state occurs.

5.3.5.1 The Min-Max algorithm for solving the two-player game tree

The below-given steps are involved in the min-max algorithm for solving the two-player game tree.

Step 1: In the first step, the algorithm generates the entire game tree and applies the utility function to get the utility values for the terminal states. In the below tree diagram, let us take A as the initial state of the tree.

Suppose maximizer takes the first turn, which has worst-case initial value = $-\infty$, and minimizer will take next turn which has worst-case initial value = $+\infty$.

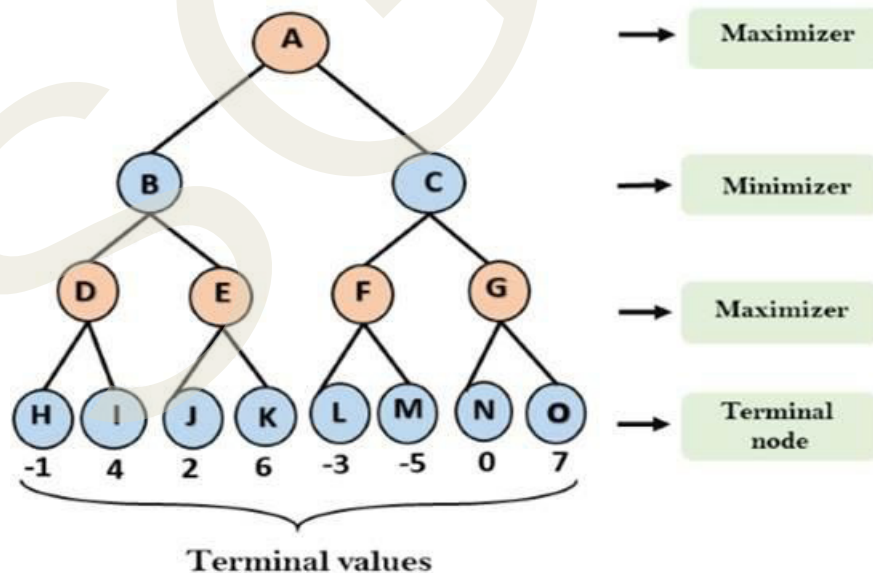


Fig 5.3.25 Initial Stage in Min-Max Algorithm

Step 2: Find the utility value for the maximizer; its initial value is $-\infty$, so compare each value in the terminal state with an initial value of the maximizer and determine

the higher nodes values. It will find the maximum among them all. For example, from Figure 4.3.26, the node value of D, E, F, and G will be 4, 6, -3, and 7, respectively.

- ◆ For node D $\max(-1, -\infty) \Rightarrow \max(-1, 4) = 4$
- ◆ For Node E $\max(2, -\infty) \Rightarrow \max(2, 6) = 6$
- ◆ For Node F $\max(-3, -\infty) \Rightarrow \max(-3, -5) = -3$
- ◆ For node G $\max(0, -\infty) = \max(0, 7) = 7$

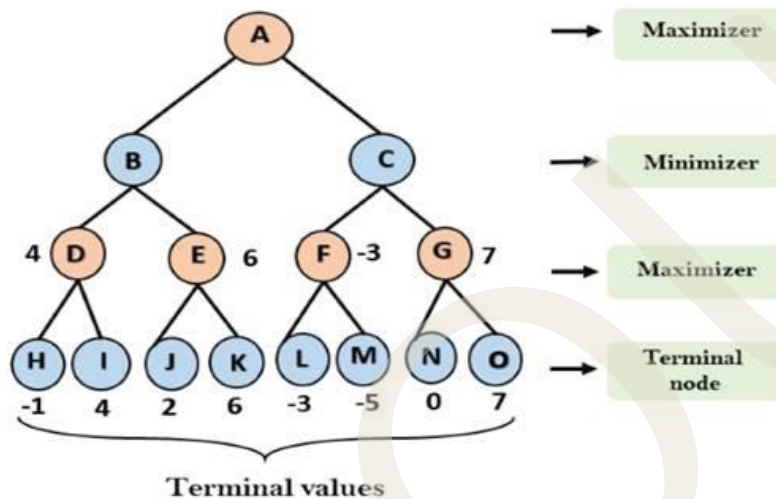


Fig 5.3.26 Second Stage in Min-Max Algorithm

Step 3: In the third step, it is a turn for minimizer, so it will compare all node values with $+\infty$ and find the 3rd layer node values.

- ◆ For node B $= \min(4, 6) = 4$
- ◆ For node C $= \min(-3, 7) = -3$

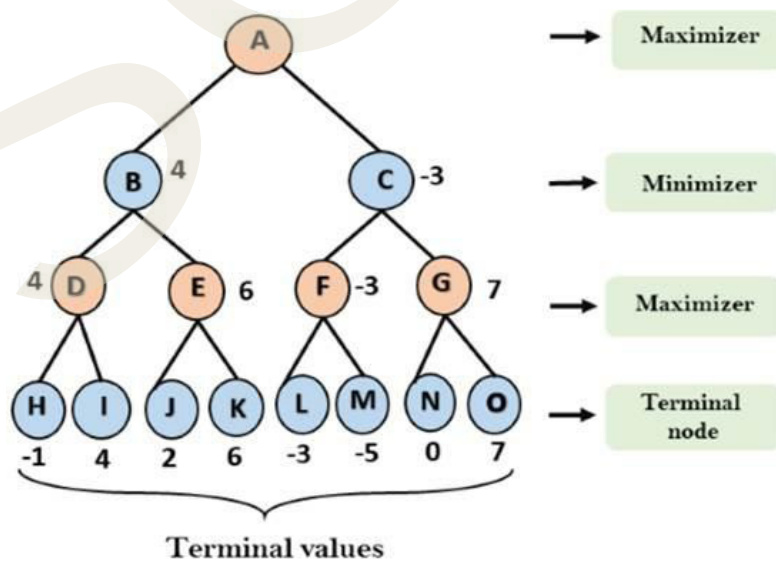


Fig 4.3.27 Third Stage in Min-Max Algorithm

Step 4: Now, it is a turn for the maximizer. Figure 4.3.28 represents the maximum of all node values, and the method finds the maximum value for the root node. In this game tree, there are only four layers. Hence, reach the root node immediately, but there will be more than four layers in actual games.

♦ For node A $\max(4, -3) = 4$

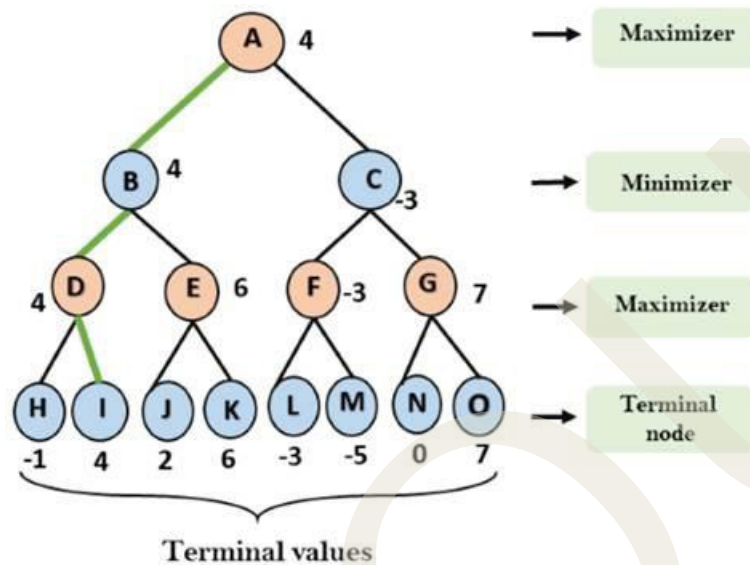


Fig 5.3.28 Fourth Stage in Min-Max Algorithm

That was the complete workflow of the mini max two-player game algorithm. The main drawback of the mini max algorithm is that it gets slow for complex games such as Chess, go, etc.

5.3.6 Matrix multiplication (Strassen's algorithm)

Matrix multiplication is a common operation in computer science and mathematics, used in areas like graphics, simulations, and machine learning. The standard method multiplies two $n \times n$ matrices in $O(n^3)$ time. Strassen's algorithm improves this by reducing the number of multiplications from 8 to 7, lowering the time to about $O(n^{2.81})$.

Need for Strassen's Algorithm

In regular matrix multiplication, every element in the resulting matrix is computed using row-by-column multiplication, requiring 8 multiplications for 2×2 matrices. Strassen discovered a method to do it with only 7 multiplications, saving time for larger matrices.

Assume matrices A and B are 2×2 matrices:

Let:

$$A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}$$

$$B = \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

1. Divide each matrix into four $(n/2) \times (n/2)$ submatrices.
2. Compute the following 7 products (instead of 8 in the conventional method):

$$M1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$M2 = (A_{21} + A_{22}) \times B_{11}$$

$$M3 = A_{11} \times (B_{12} - B_{22})$$

$$M4 = A_{22} \times (B_{21} - B_{11})$$

$$M5 = (A_{11} + A_{12}) \times B_{22}$$

$$M6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$

$$M7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

3. Use these products to compute the four parts of the result matrix **C**:

$$C_{11} = M1 + M4 - M5 + M7$$

$$C_{12} = M3 + M5$$

$$C_{21} = M2 + M4$$

$$C_{22} = M1 - M2 + M3 + M6$$

4. Combine the four **C** submatrices to form the final matrix **C**.

Matrix Division

Strassen's algorithm starts by dividing each input matrix into 4 equal submatrices.

Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow a = 1, b = 2, c = 3, d = 4$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \rightarrow e = 5, f = 6, g = 7, h = 8$$

Strassen's 7 Multiplications

Instead of doing 8 direct multiplications, we calculate 7 intermediate matrices:

$$M1 = (a + d)(e + h) = (1+4)(5+8) = 5*13 = 65$$

$$M2 = (c + d)e = (3+4)*5 = 7*5 = 35$$

$$M3 = a(f - h) = 1*(6-8) = -2$$

$$M4 = d(g - e) = 4*(7-5) = 4*2 = 8$$

$$M5 = (a + b)h = (1+2)*8 = 3*8 = 24$$

$$M6 = (c - a)(e + f) = (3-1)*(5+6) = 2*11 = 22$$

$$M7 = (b - d)(g + h) = (2-4)*(7+8) = -2*15 = -30$$

Using M1 to M7, we build the final result matrix:

$$C11 = M1 + M4 - M5 + M7 = 65 + 8 - 24 - 30 = 19$$

$$C12 = M3 + M5 = -2 + 24 = 22$$

$$C21 = M2 + M4 = 35 + 8 = 43$$

$$C22 = M1 - M2 + M3 + M6 = 65 - 35 - 2 + 22 = 50$$

So, $C = [[19, 22], [43, 50]]$

Recursive Strategy

For larger matrices (e.g., 4x4, 8x8), we apply the same process recursively by dividing each matrix into 2x2 blocks and applying Strassen's method to those smaller parts.

Time Complexity

- ◆ Standard method: $O(n^3)$
- ◆ Strassen's method: $O(n^{\log_2 7}) \approx O(n^{2.81})$
- ◆ This reduction becomes significant for large matrix sizes.

Advantages

- ◆ Fewer multiplications (7 vs 8)
- ◆ Faster for large matrices
- ◆ Useful in big data and scientific computation

Disadvantages

- ◆ Involves more additions/subtractions
- ◆ Complex logic
- ◆ Not efficient for small matrices

Applications

- ◆ 3D Computer Graphics
- ◆ Machine Learning (Matrix operations in neural networks)
- ◆ Simulations and Scientific Computing

Recap

- ◆ Divide and Conquer Algorithm: This algorithmic pattern solves problems by recursively breaking them into smaller sub-problems, solving those, and then combining their solutions.
- ◆ Binary Search Algorithm: An efficient search technique that works exclusively on sorted lists by repeatedly dividing the search interval in half.
- ◆ Merge Sort: A sorting algorithm that applies the divide-and-conquer strategy to sort an array by recursively splitting it into single-element sub-arrays and then merging them back in sorted order.
- ◆ Quick Sort Algorithm: A highly efficient sorting algorithm that uses a divide-and-conquer approach to partition an array around a “pivot” element and then recursively sorts the sub-arrays.
- ◆ Min-max Algorithm: A decision-making algorithm, primarily used in turn-based, two-player games, that determines the optimal move for a player by assuming the opponent also plays optimally.
- ◆ Matrix multiplication (Strassen’s algorithm): An optimized algorithm that improves upon standard matrix multiplication by reducing the number of arithmetic operations, particularly beneficial for large matrices.

Objective Type Questions

1. What is the core principle behind the Divide and Conquer algorithmic paradigm?
2. Name the three main parts or steps involved in a general Divide and Conquer algorithm.
3. Which specific data structure is required for the Binary Search Algorithm to work efficiently?
4. In the context of Quick Sort, what is the role of the “pivot” element?
5. What is the primary goal of the Min-max algorithm in a two-player game?
6. How does Strassen’s algorithm improve upon the standard method of matrix multiplication for large matrices?
7. What is the worst-case time complexity of Merge Sort?
8. Mention one significant advantage of the Divide and Conquer approach.
9. Mention one significant disadvantage of the Divide and Conquer approach.
10. Apart from sorting, name one other problem type where the Divide and Conquer strategy is applied.

Answers to Objective Type Questions

1. Recursion
2. Divide, Conquer, Combine
3. Array
4. Partitioning
5. Optimization
6. Fewer multiplications
7. $O(n \log n)$
8. Efficiency
9. Overhead
10. Searching

Assignments

1. Explain the core principle of the Divide and Conquer algorithmic paradigm in your own words. Describe its three fundamental steps: Divide, Conquer, and Combine.
2. Discuss the advantages and disadvantages of using the Divide and Conquer approach in algorithm design. Provide at least three points for each.
3. What are the two fundamental concepts in the divide and conquer strategy? Define each in brief.
4. Provide examples of at least three common algorithms that utilize the Divide and Conquer strategy.

Specific Algorithms

5. Binary Search Algorithm:

Explain why a sorted list is a prerequisite for the Binary Search Algorithm.

Trace the steps of the Binary Search Algorithm to find the element 20 in the array: [5, 10, 15, 20, 25, 30, 35, 40]. Show each comparison and the sub-list being considered.

6. Merge Sort:

Describe the “Divide” and “Conquer/Combine” phases of the Merge Sort algorithm.

Sort the following array using Merge Sort, illustrating each step of the division and merging process: [11, 5, 13, 7, 2, 9]

7. Quick Sort:

Explain the role of the “pivot” element in Quick Sort and how it contributes to the partitioning process.

Trace the Quick Sort algorithm for the following array, assuming the first element is always chosen as the pivot: [7, 2, 9, 1, 5, 8] Show the array state after each partition.

Reference

1. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Python* (1st ed.). Wiley. [Pragmatic Bookshelf+5wiley.com+5Reddit+5](#) (This title remains current; a “zyBooks” digital version is also available.) [zyBooks+1Reddit+1](#)
2. A Common-Sense Guide to Data Structures and Algorithms in Python (2nd ed.) by Jay Wengrow (2022). (You can use this as a more accessible companion to the above texts.) [pythonbooks.org+7Reddit+7Pragmatic Bookshelf+7](#)
3. Knuth, D. E. (2023). *The Art of Computer Programming, Volumes 1–4B Boxed Set*. Addison-Wesley. [Wikipedia](#)
4. Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Prentice Hall.

Suggested Reading

1. GeeksforGeeks. (n.d.). *Data structures*. <https://www.geeksforgeeks.org/data-structures>
2. Programiz. (n.d.). *Data structures tutorial*. <https://www.programiz.com/dsa>
3. TutorialsPoint. (n.d.). *Data structures and algorithms tutorial*. https://www.tutorialspoint.com/data_structures_algorithms
4. National Programme on Technology Enhanced Learning (NPTEL). (n.d.). *Data structures and algorithms*. <https://nptel.ac.in>

Unit 4

Minimum Cost Spanning Trees

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- ♦ define spanning tree and minimum spanning tree.
- ♦ state the difference between Prim's and Kruskal's algorithms with examples.
- ♦ identify the steps of Prim's and Kruskal's algorithms to find a minimum spanning tree.
- ♦ list the advantages and limitations of Prim's and Kruskal's algorithms for different types of graphs.

Prerequisites

Suppose a cable TV service provider wants to connect multiple homes in a newly developed residential area. The objective is to lay out the cables in such a way that every house is connected, but the total cost of cabling is minimized. Importantly, there should be no loops or redundant connections. This real-time problem of connecting different locations with the least amount of resources directly relates to the concept of spanning trees and minimum cost spanning trees in computer science. Similar challenges arise when designing roads, water supply systems, or even internet routing networks, where efficient resource usage is critical. These scenarios help demonstrate the relevance of learning about spanning trees in both academic and practical domains.

In the context of graph theory, a spanning tree is a subset of the edges of a connected, undirected graph that connects all the vertices together without any cycles and with the minimum possible number of edges (exactly one less than the number of vertices). A minimum spanning tree (MST) is a spanning tree with the smallest possible total edge weight among all spanning trees of a graph. To find such trees efficiently, we use algorithms like Kruskal's algorithm and Prim's algorithm, which are based on greedy strategies. These algorithms not only offer optimized solutions but also demonstrate essential concepts such as edge selection, cycle detection, and tree construction. Understanding and studying minimum cost spanning trees provides several benefits. It enhances a learner's ability to solve real-life optimization problems where cost, time, and resources. Moreover, this topic strengthens the learner's grasp of graph-based problem-solving techniques, which are fundamental in various fields including operations



research, transportation planning, and computer networking. By integrating theoretical knowledge with hands-on learning and relatable applications, students are more likely to grasp the significance of minimum cost spanning trees and develop both interest and competence in algorithmic problem solving.

Key words

Concept of Spanning Trees, Minimum Spanning Tree (MST), Prim's Algorithm, Kruskal's Algorithm

Discussion

5.4.1 Spanning Trees

Suppose you are a cable tv network distributor and you have to give new connections in a particular residential area. Let's assume that area includes 10 houses. You are going to give connections to 10 houses in that area. Fig. 4.4.1 shows the map of that residential area with different routes. With the help of this map, we can easily get an idea about how to connect these 10 houses in different ways. For this, let us assume each house as a node. If a connecting path between 2 houses exists, then we can consider it as an edge between those 2 nodes (houses).

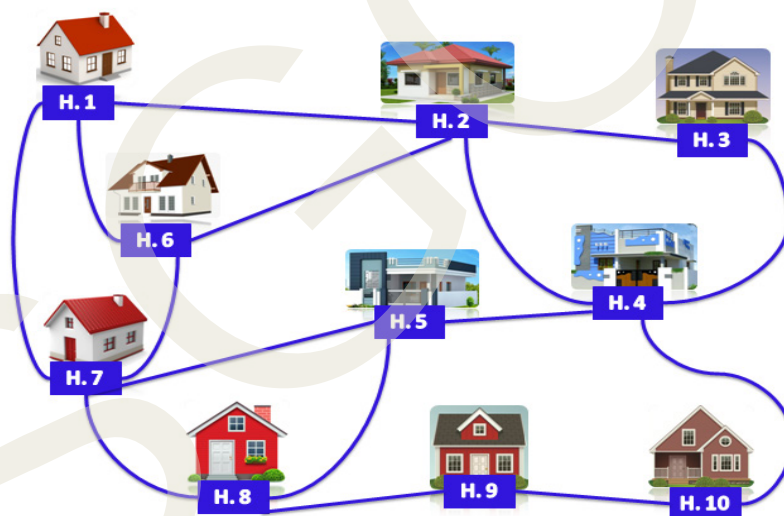


Fig 5.4.1 A Particular Residential Area Route Map

After constructing all the possible ways, we select one of the best ways (route) for giving connection to all the houses. While constructing different routes we have to consider several things. The main thing is that we must include all the houses in that area. No looping paths are included because it wastes your time and money. So, you have to reject some paths that will create a cycle or loop. Basically, you are constructing a tree that spans all the nodes of a given graph. Here the houses and the routes are represented with nodes and edges of a graph. The tree is acyclic and contains all the nodes of the graph. We can call it a spanning tree of that graph.

A spanning tree of a graph is just a sub graph that contains all the vertices. It is a tree. If G is a connected graph and its spanning tree contains all the vertices and the edges which connect all the vertices. So the number of edges will be 1 less than the number of nodes. A graph may have many spanning trees. The resultant graph obtained by BFS and DFS is a spanning tree. Spanning tree is basically used to find a minimum path to connect all nodes in a graph.

Properties

- ◆ A spanning tree doesn't have cycles and it cannot be disconnected.
- ◆ Every connected and undirected graph has at least one spanning tree.
- ◆ A disconnected graph doesn't have any spanning tree as it cannot be spanned to all its vertices.
- ◆ A complete undirected graph can have a maximum n^{n-2} number of spanning trees, where n is the number of nodes.
- ◆ Removing one edge from the spanning tree will make the graph disconnected.
- ◆ Adding one edge to the spanning tree will create a circuit or loop.

If a disconnected graph with n vertices has edges less than $n-1$ then no spanning tree is possible. The applications of spanning trees include the areas of civil network planning (water supply, telephone lines, electric lines), cluster analysis, computer networks routing protocols etc. Fig. 5.4.2 shows a connected graph G and its possible spanning trees. Here we can see the total number of edges in each spanning tree is 1 less than the number of nodes. (Number of nodes = 4, Number of edges in each spanning tree = 3)

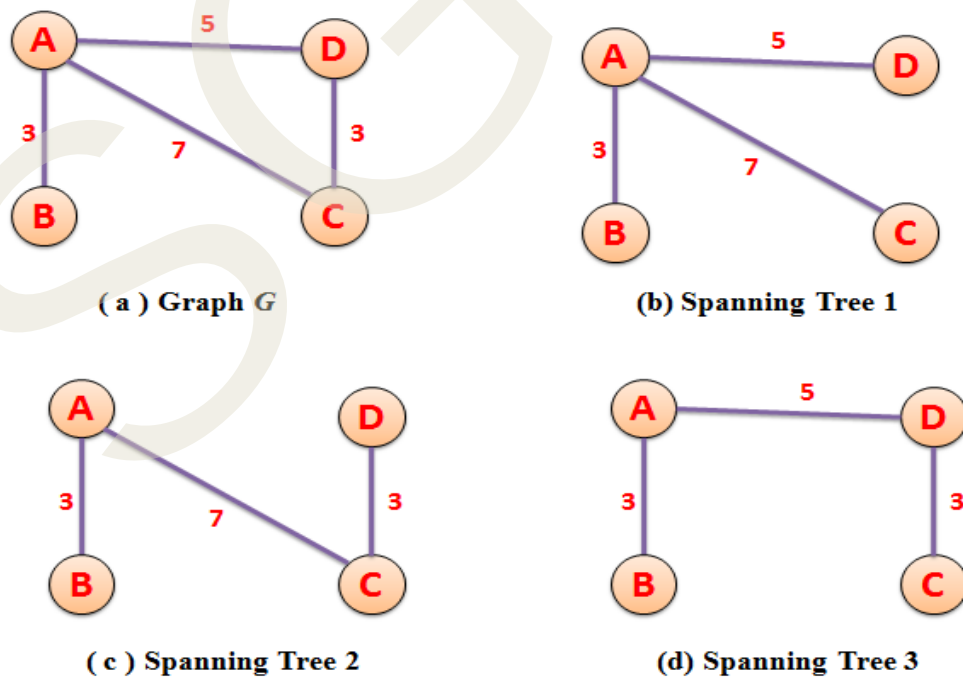


Fig 5.4.2 Spanning Trees.

5.4.2 Minimum Spanning Tree (MST)

Consider the previous example of cable tv network distribution. We have to consider a possible network of routes that minimize the overall cost. That means our task is to set up lines in such a way that all the houses are connected and the cost of setting up the whole connection is minimal.

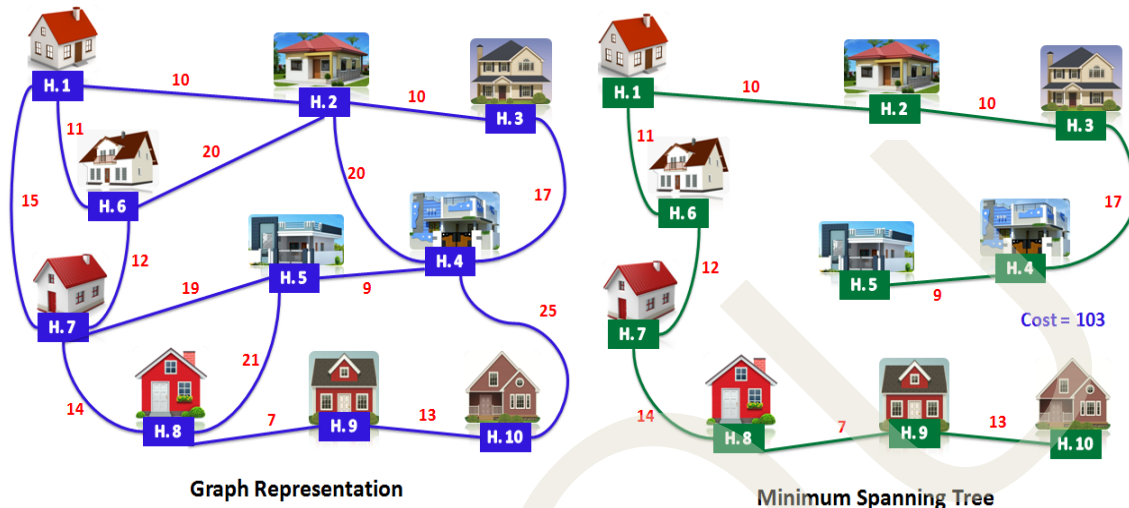


Fig 5.4.3 Cable TV Network example graph and its minimum spanning tree.

Fig. 5.4.3 shows the graph representation of this problem. Here each node represents a house and each weighted edge represents the cost of setting up the connection between the corresponding two houses. For example, house 1 and house 2 are connected with an edge having weight 10. This means that for setting up a connection line between house 1 and house 2 we need a total cost 10. So our aim is to reduce the total cost of the entire connection. In Fig. 5.4.3, we can see that all the houses are connected and the cost of setting up the whole connection is minimized. That is the sum of all the edges is minimum (here minimum cost is 103). We can call it a minimum cost spanning tree or simply minimum spanning tree (MST).

Given a connected weighted graph G , a spanning tree T is created such that the sum of the weights of the tree edges in T is as small as possible is known as the minimum spanning tree. The cost of a spanning tree is the sum of costs of the edges in the tree.

Fig. 5.4.4(a) shows another example of a weighted graph with 4 vertices and 4 edges. We can see all the three possible spanning trees of the given graph in Fig. 5.4.4(a). Fig. 5.4.4(b) shows the first spanning tree with a cost 15 and Fig. 5.4.4(c) shows the second spanning tree with a cost 13. Fig. 5.4.4(d) is the third spanning tree with a cost 11. Fig. 5.4.4(d) shows the minimum spanning tree of the given graph G because the sum of the weights of edges in 5.4.4(d) is 11. Sum of the weights of edges in Fig 5.4.4(b) is 15 and in Fig 5.4.4(c) is 13. So the minimum spanning tree is Fig 5.4.4(d).

A typical application for minimum cost spanning trees occurs in the design of telecommunications networks. The vertices of a graph represent cities and the edges of possible communications links between the cities. The cost associated with an edge

represents the cost of selecting that link for the network. A minimum cost spanning tree represents a communications network that connects all the cities at minimal cost. Minimum spanning trees are also useful in finding the paths in the map. For designing networks like water supply networks and electrical grids we can use the idea of minimum spanning tree.

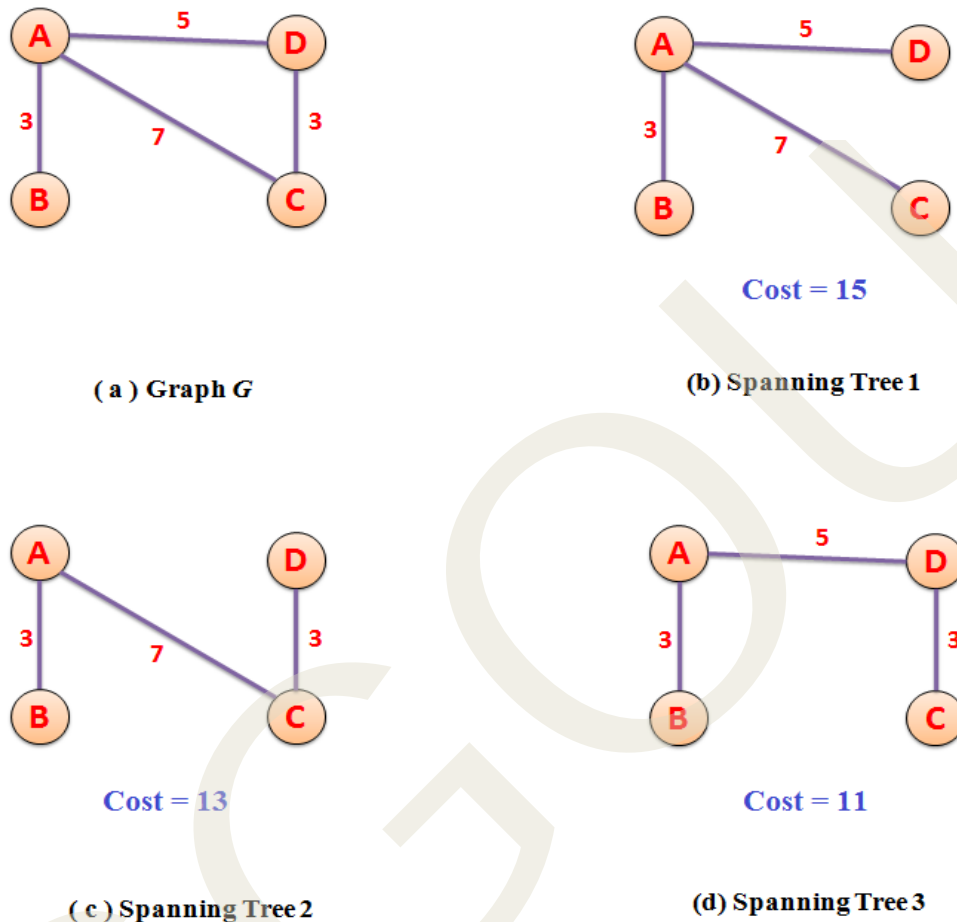


Fig 5.4.4 Minimum Spanning Tree Example.

5.4.3 Prim's Algorithm

Prim's algorithm is used to find the minimum spanning tree of a given graph. We can apply this algorithm on a graph which must be weighted, connected and undirected. This algorithm finds a subset of the edges that forms a tree that includes every node. Here the total weight of all the edges in the tree is minimized. In Prim's algorithm the minimum spanning tree grows naturally, starting from an arbitrary root. At each stage, we add a new branch to the tree already constructed and the algorithm stops when all the nodes have been reached.

Algorithm

Step 1 : Randomly choose any node as the starting node and include that node in the spanning tree.

Step 2 : Find all the edges that connect the spanning tree to the new nodes and insert them into the incident edge set.

Step 3 : Find the least weight edge among the edges in the given incident edge set.

Step 4 : Check whether including that edge in the existing spanning tree forms a cycle or not.

Step 5 : If including that edge creates no cycle, then add that edge to the spanning tree. Otherwise, reject that edge from the incident edge set and go to step 3.

Step 6 : Repeat steps 2 to 5 until all the nodes are included in the spanning tree and the minimum spanning tree is obtained.

Example

Consider an example graph shown in Fig. 5.4.5. Now we can see how Prim's algorithm works on the given graph. The given graph contains 7 nodes and 12 weighted edges.

Step 1:

Choose any node from the graph randomly. Here we choose node "A" as a starting node. Then it is included in the spanning tree. Fig. 5.4.6 shows the resultant graph and spanning tree.

Step 2:

Now find the edges connecting the new adjacent nodes of A. that is edges from node A to the new adjacent nodes B and D. Here 2 edges are there with weights 1 and 5. Insert them into the incident edge set. Now find the least weight edge from this set. Here (A, B) is the least weight edge and it doesn't form any cycle. So include the edge (A, B) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 5.4.7.

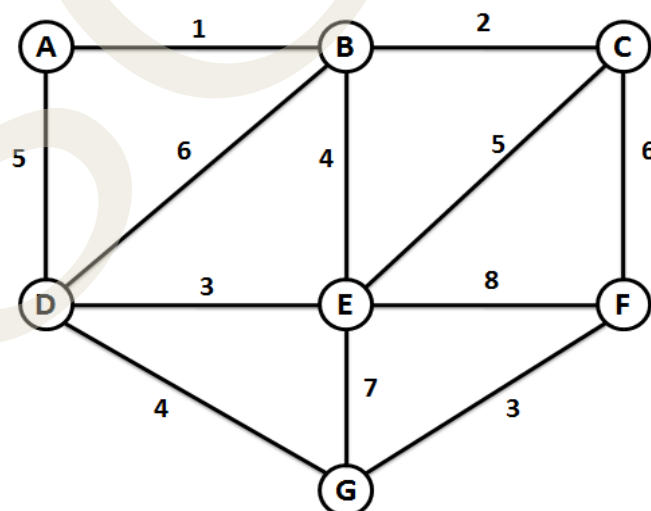


Fig 5.4.5 Example graph for Prim's algorithm

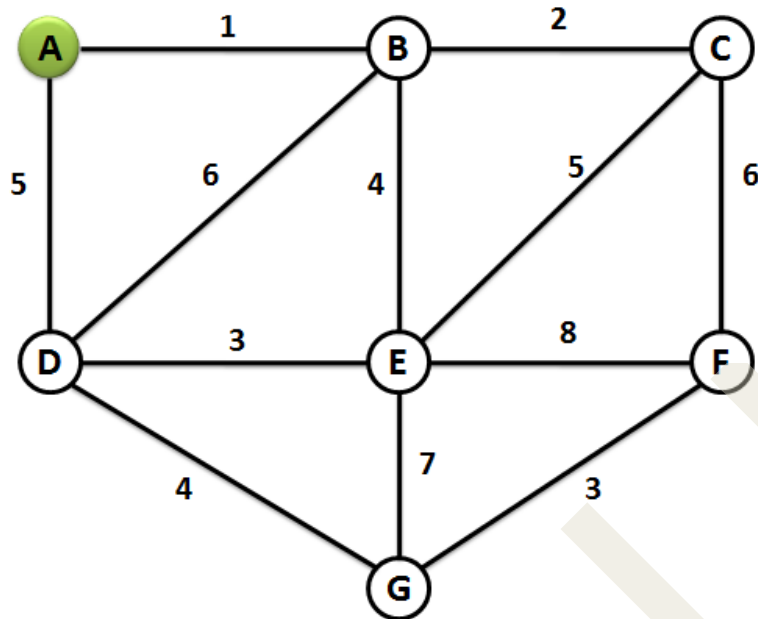


Fig 5.4.6 Prim's Algorithm – Resultant Graph, Spanning Tree (1).

Step 3:

Now find the edges connecting the new adjacent nodes of A and B. That is edges from nodes A and B to the new adjacent nodes C, D and E. Here 4 edges are there with weights 5, 6, 4 and 2. That is edges (A, D), (B, D), (B, E), and (B, C). Insert them into the incident edge set. Now find the least weight edge from this set. Here (B, C) is the least weight edge and including this edge to the spanning tree, doesn't form any cycle. So include the edge (B, C) to the spanning tree. The resultant graph and spanning tree is shown in Fig. 5.4.8.

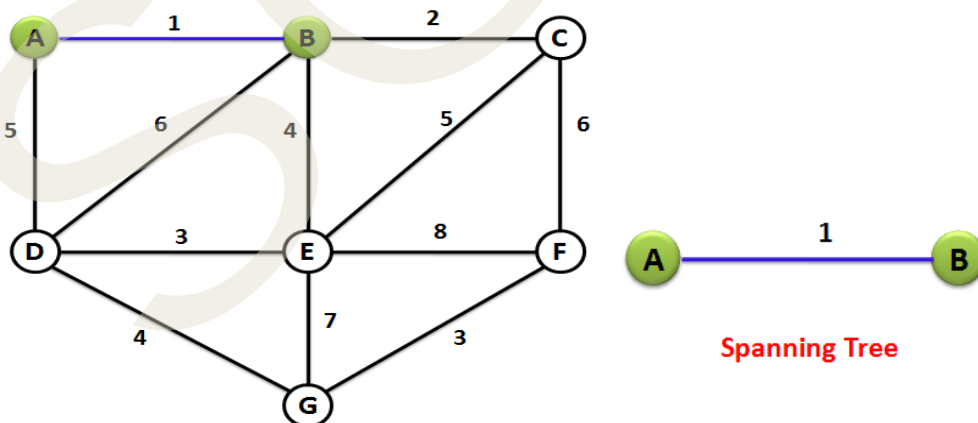


Fig 5.4.7 Prim's Algorithm – Resultant Graph, Spanning Tree (2).

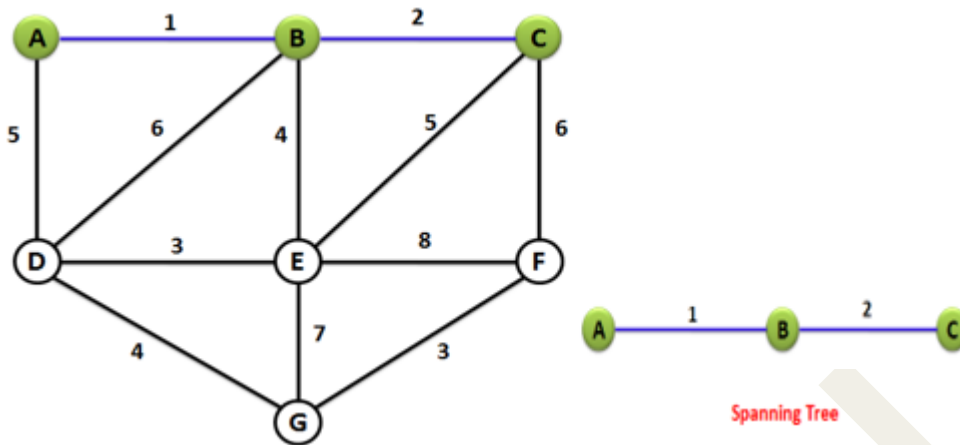


Fig 5.4.8 Prim's Algorithm – Resultant Graph, Spanning Tree (3)

Step 4:

Now find the edges connecting the new adjacent nodes of A, B, and C. That is edges from the nodes A, B, and C to the new adjacent nodes D, E and F. Here 5 edges are there with weights 5, 6, 4, 5 and 6. That is edges (A, D), (B, D), (B, E), (C, E) and (C, F). Insert them into the incident edge set. Now find the least weight edge from this set. Here (B, E) is the least weight edge and including this edge to the spanning tree, doesn't form any cycle. So include the edge (B, E) to the spanning tree. The resultant graph and spanning tree is shown in Fig. 5.4.9.

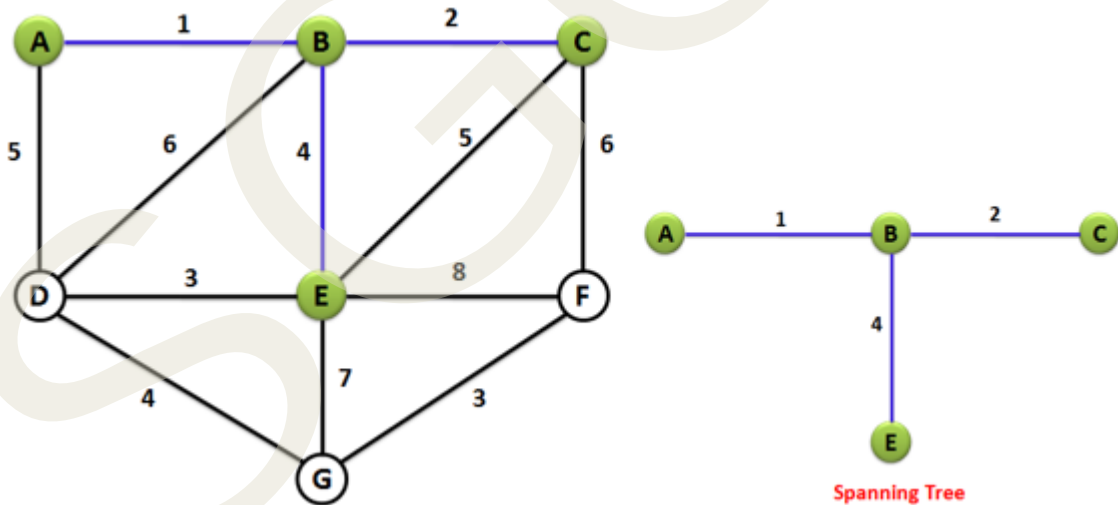


Fig 5.4.9 Prim's Algorithm – Resultant Graph, Spanning Tree (4)

Step 5:

Now find the edges connecting the new adjacent nodes of A, B, C, and E. That is edges from the nodes A, B, C and E to the new adjacent nodes D, F and G. Here 6 edges are there with weights 5, 6, 6, 3, 7 and 8. That is edges (A, D), (B, D), (C, F), (E, D), (E, G) and (E, F). Insert these 5 edges into the incident edge set. Now find the least weight edge from this set. Here (E, D) is the least weight edge and including this edge to the

spanning tree, doesn't form any cycle. So include the edge (E, D) to the spanning tree. The resultant graph and spanning tree is shown in Fig. 5.4.10.

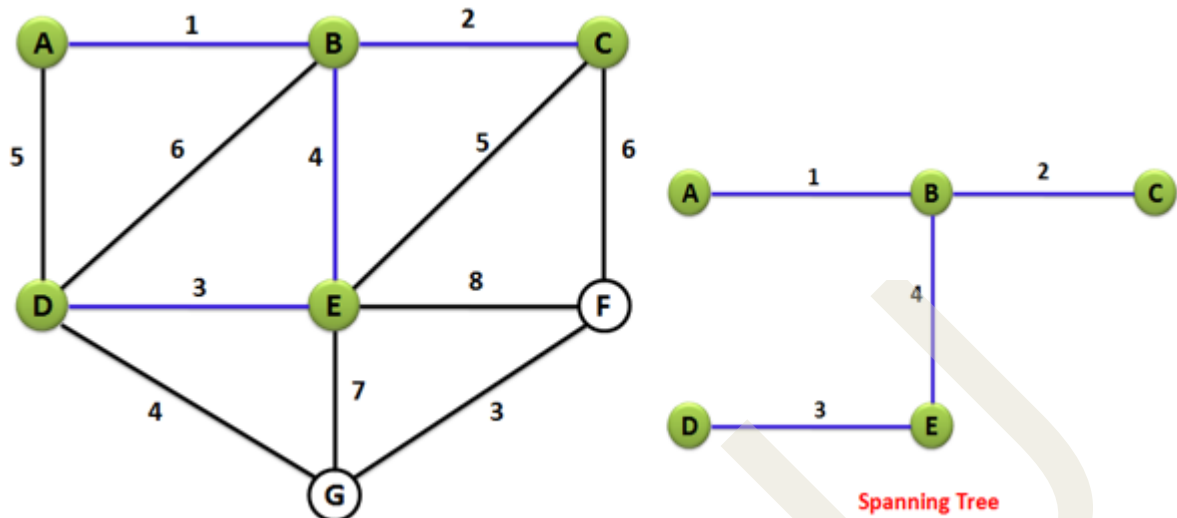


Fig 5.4.10: Prim's Algorithm – Resultant Graph, Spanning Tree (5)

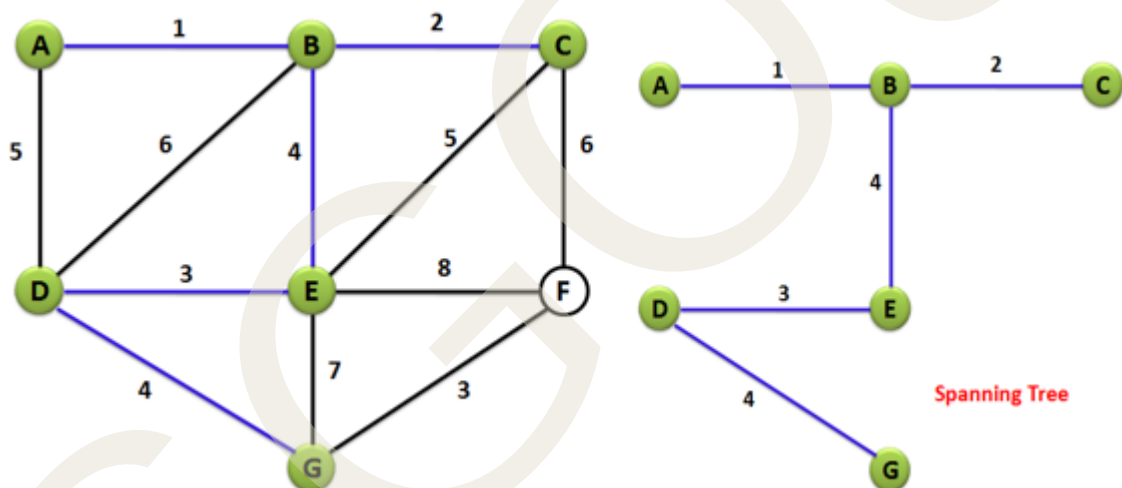


Fig 5.4.11 Prim's Algorithm – Resultant Graph, Spanning Tree (6).

Step 6:

There is no new adjacent nodes from A and B. So now we have to find the edges connecting the new adjacent nodes of C, E and D. That is edges from the nodes C, E, and D to the new adjacent nodes G and F. Here 4 edges are there with weights 6, 8, 7 and 4. That is edges (C, F), (E, F), (E, G) and (D, G). Insert these 4 edges into the incident edge set.

Now find the least weight edge from this set. Here (D, G) is the least weight edge and including this edge to the spanning tree, doesn't form any cycle. So include the edge (D, G) to the spanning tree. The resultant graph and spanning tree is shown in Fig. 5.4.11.

Step 7:

There is no new adjacent nodes from A, B and D. So now we have to find the edges

connecting the new adjacent nodes of C, E and G. That is edges from the nodes C, E, and G to the new adjacent node F. Here 3 edges are there with weights 6, 8, and 3. That is edges (C, F), (E, F), and (G, F). Insert these 3 edges into the incident edge set. Now find the least weight edge from this set. Here (G, F) is the least weight edge and including this edge to the spanning tree, doesn't form any cycle. So include the edge (G, F) to the spanning tree. The resultant graph and its minimum spanning tree are shown in Fig. 5.4.12.

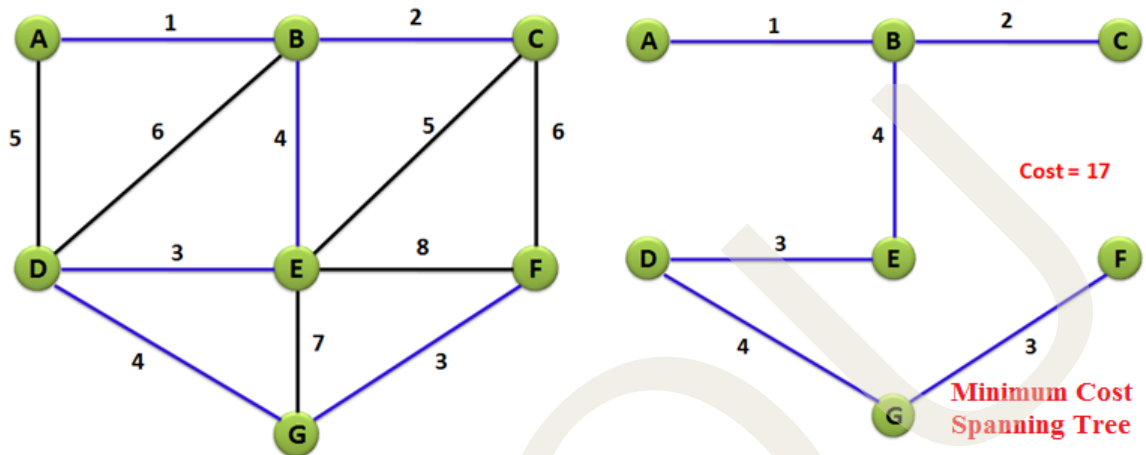


Fig. 5.4.12 Prim's Algorithm – Resultant Graph, Minimum Spanning Tree

After performing Prim's algorithm on an undirected, weighted and connected graph, the resultant spanning tree formed will be the minimum cost spanning tree. The cost of a spanning tree is the sum of the weights of all the edges present in that spanning tree. Here 6 edges are there in the spanning tree. They are (A, B), (B, C), (B, E), (E, D), (D, G) and (G, F). The sum of the weights of these six edges is 17. So the minimum cost is 17.

5.4.3.1 Applications of Prim's Algorithm

- ◆ Network Design: Helps in building computer networks or telecom systems with minimum cabling cost.
- ◆ Road and Railway Construction: Used to plan paths that connect all locations using the shortest possible total route length.
- ◆ Cable TV and Utility Grid Layouts: Assists utility providers in laying cables or pipelines to serve every home with minimum resources.
- ◆ Cluster Analysis: Supports hierarchical clustering methods in data mining by connecting data points with minimal total distance.
- ◆ Travelling Salesman Approximation: Forms a base for approximation algorithms to solve hard problems like TSP.

5.4.4 Kruskal's Algorithm

Kruskal's algorithm creates a forest of trees. Initially, the forest contains n single node

trees and no edges. At each step we add one edge so that it joins two trees together. If it creates a cycle then that edge is not included in the tree. In this method, we first examine all the edges and sort them in the increasing order of their weights. Then it is stored in a priority queue. That is the first priority that goes to the edge with the least value.

Thus the edges are examined one by one according to their weights and decide whether the selected edge should be included in the spanning tree or not. If the two nodes of the selected edge belong to the same tree then we will not include that edge in the spanning tree. Because the two nodes are in the same tree and they are already connected. Adding this edge will create a cycle. So we simply reject that edge from the spanning tree. We will insert an edge in the spanning tree only if its nodes are in different trees.

Algorithm

Step 1: Initially construct a separate tree for each node in a graph.

Step 2: Sort the edges in ascending order according to their weights.

Step 3: Store the sorted edges in a priority queue.

Step 4: Examine all the edges one by one from the priority queue.

Step 4.1 Check whether including the edge will create a cycle or not.

Step 4.2 If it creates no cycle and then adds that edge to the spanning tree,

Otherwise reject that edge.

Example

Consider an example graph shown in Fig. 5.4.13. Now we can find the minimum cost spanning tree of the following graph using Kruskal's algorithm. The given graph contains 7 nodes and 12 weighted edges. We have to sort the edges in ascending order, according to their weights and store them in a priority queue.

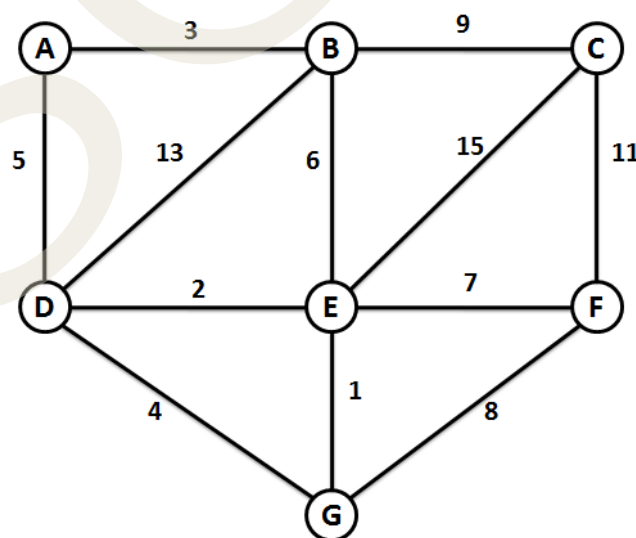


Fig. 5.4.13 Example graph for Kruskal's algorithm

Step 1:

Initially construct a separate tree for each node in a graph. Sort the edges in ascending order according to their weights and store them in a priority queue. Fig. 5.4.14 shows the resultant forest of trees and the priority queue.

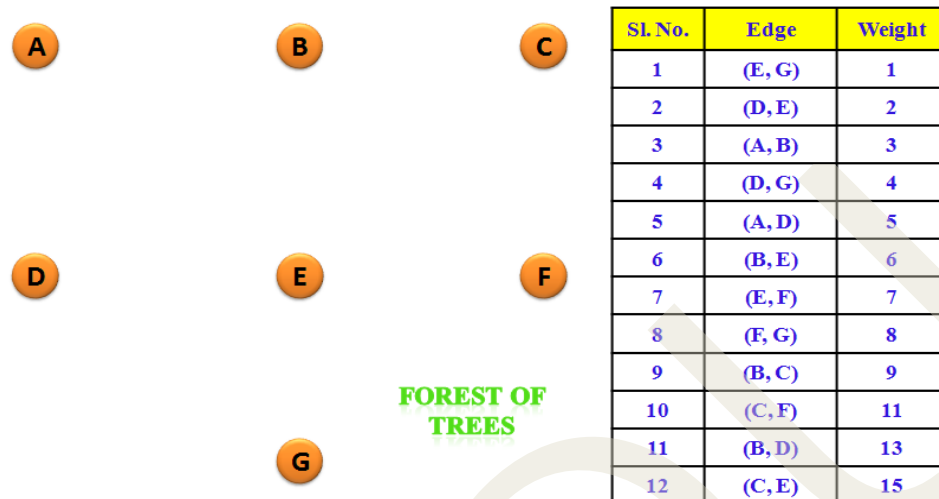


Fig 5.4.14 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (1)

Step 2:

Now examine the edges one by one from the priority queue. The least weight edge is (E, G) which is the first entry of the priority queue. Its weight is 1. Check whether including the edge (E, G) to the tree will create a cycle or not. It will not create a cycle. So include (E, G) to the tree. Fig. 5.4.15 shows the resultant forest of trees and the priority queue.

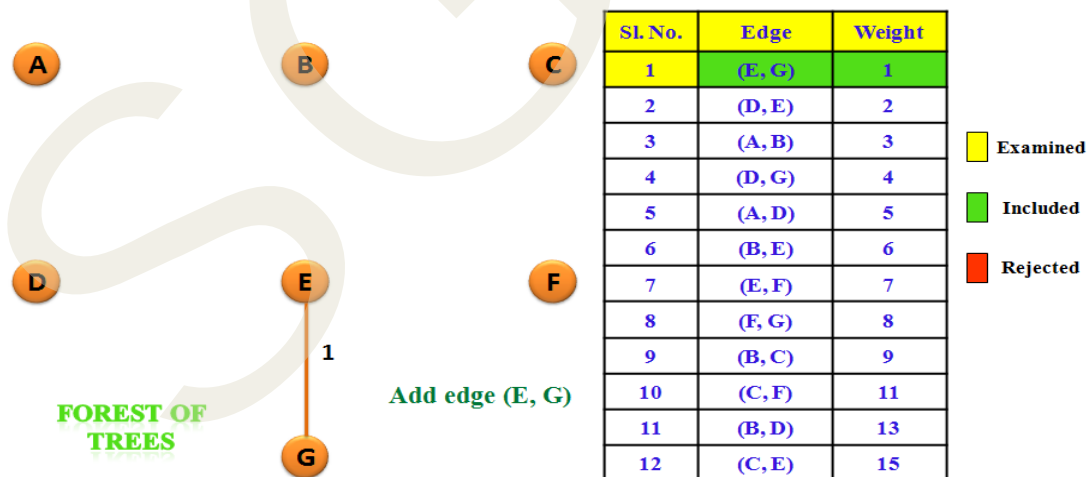


Fig 5.4.15 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (2)

Step 3:

The next edge is (D, E) with a weight 2. Check whether including the edge (D, E) to the

tree will create a cycle or not. It will not create a cycle. So include (D, E) to the tree. Fig. 5.4.16 shows the resultant forest of trees and the priority queue.

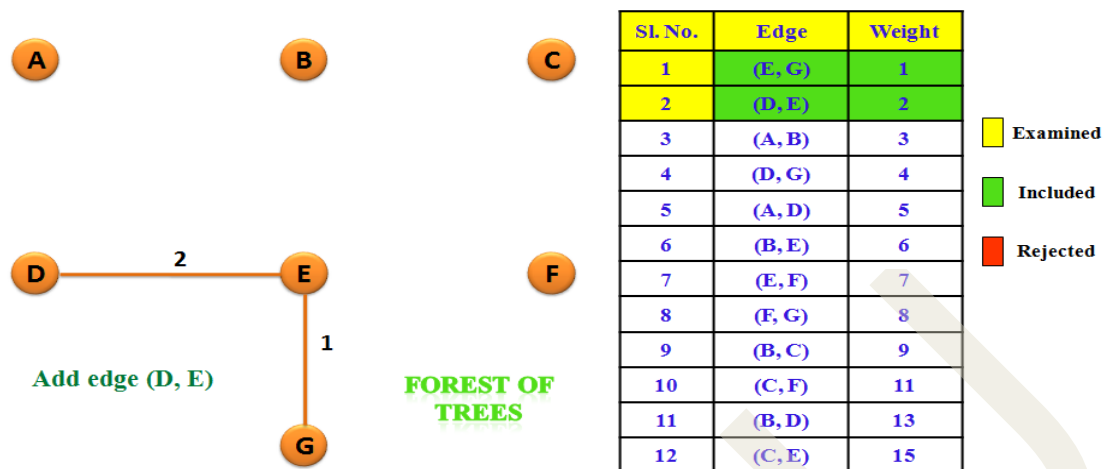


Fig 5.4.16: Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (3)

Step 4:

The next edge is (A, B) with a weight 3. Check whether including the edge (A, B) to the tree will create a cycle or not. It will not create a cycle. So include (A, B) to the tree. Fig. 5.4.17 shows the resultant forest of trees and the priority queue.

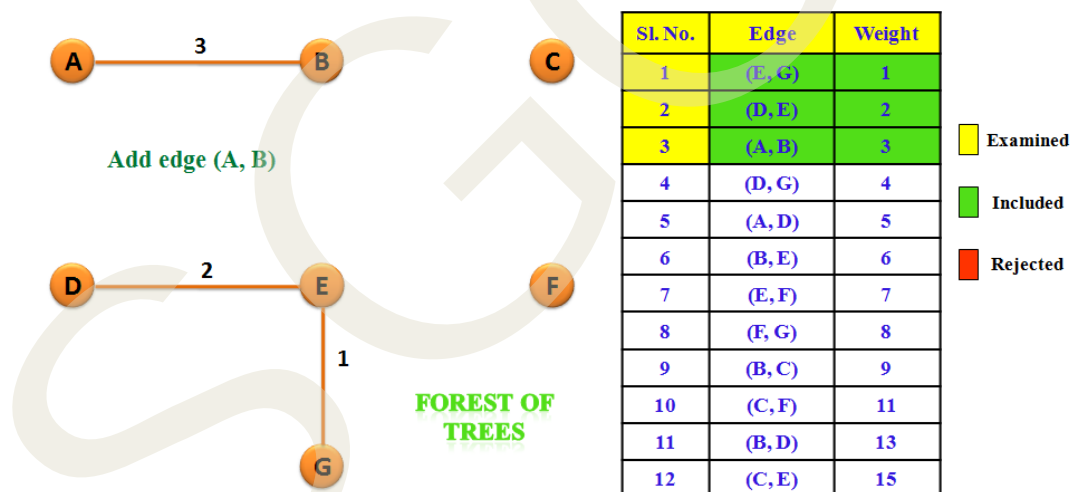


Fig 5.4.17 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (4)

Step 5:

The next edge is (D, G) with a weight 4. Check whether including the edge (D, G) to the tree will create a cycle or not. It will create a cycle. So reject (D, G) from the tree. Fig. 5.4.18 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (D, G). We do not include this edge to the tree.

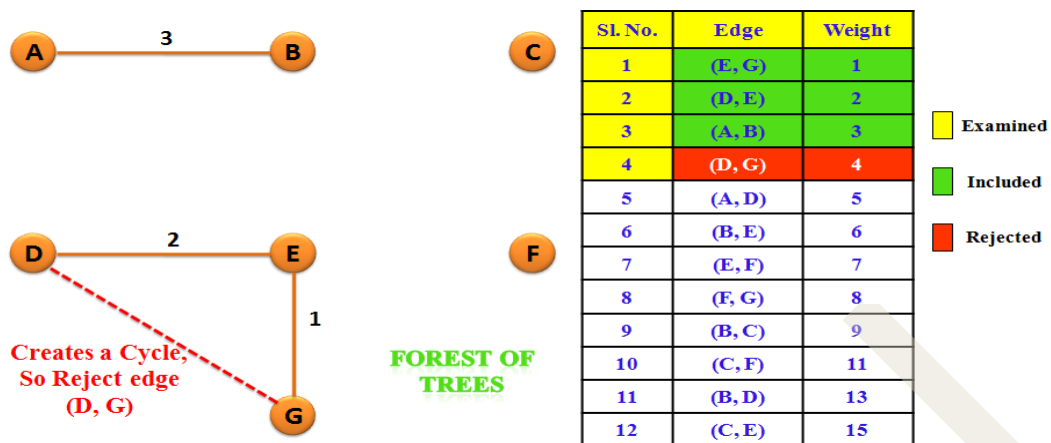


Fig 5.4.18 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (5)

Step 6:

The next edge is (A, D) with a weight 5. Check whether including the edge (A, D) to the tree will create a cycle or not. It will not create a cycle. So include (A, D) to the tree. Fig. 5.4.19 shows the resultant forest of trees and the priority queue.

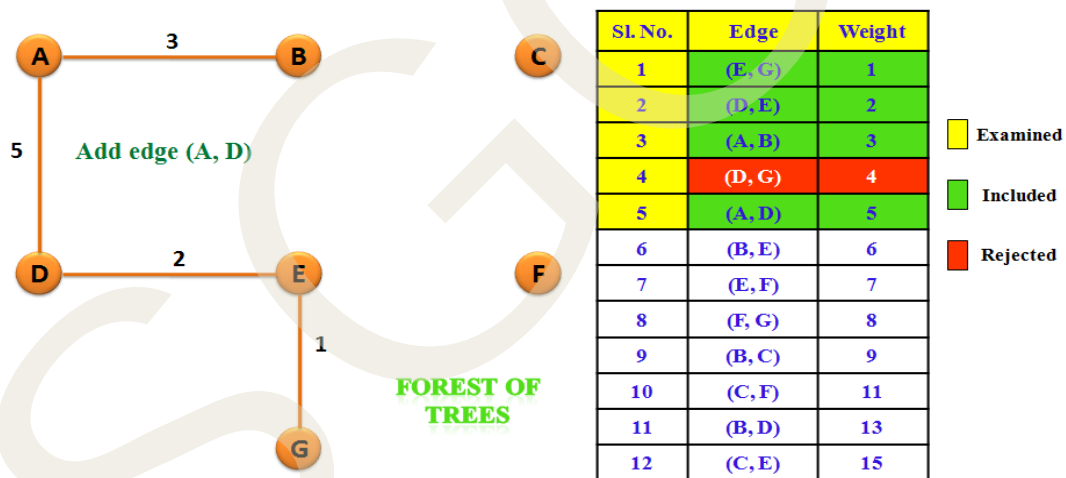


Fig 5.4.19 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (6)

Step 7:

The next edge is (B, E) with a weight 6. Check whether including the edge (B, E) to the tree will create a cycle or not. It will create a cycle. So reject the edge (B, E) from the tree. Fig. 5.4.20 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (B, E). We do not include this edge to the tree.

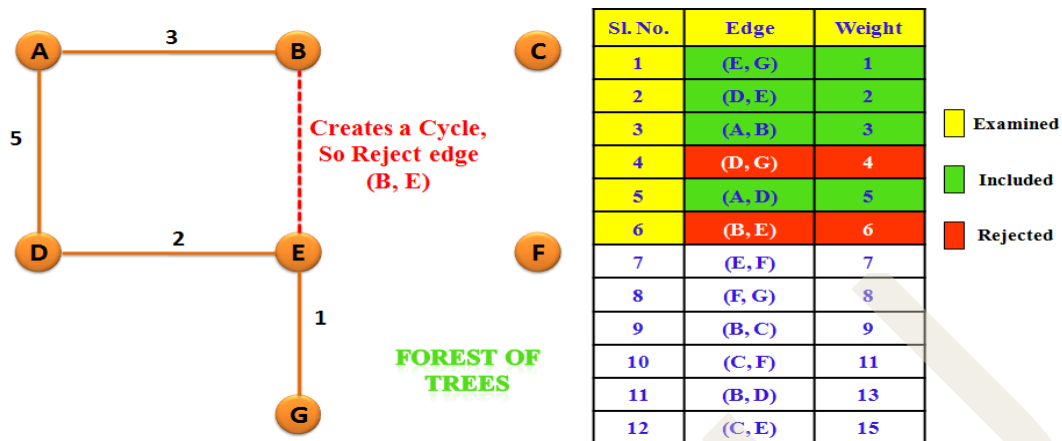


Fig 5.4.20 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (7)

Step 8:

The next edge is (E, F) with a weight of 7. Check whether including the edge (E, F) to the tree will create a cycle or not. It will not create a cycle. So include (E, F) to the tree. Fig. 5.4.21 shows the resultant forest of trees and the priority queue.

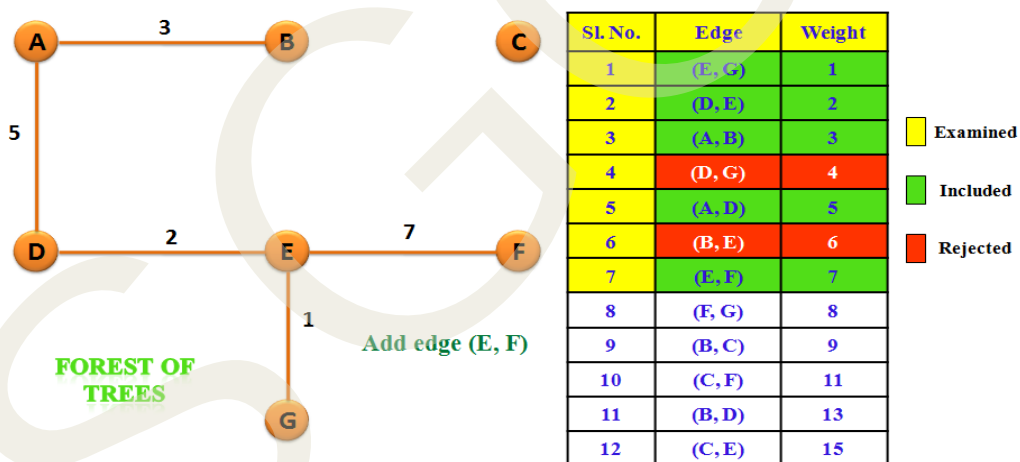


Fig 5.4.21 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (8)

Step 9:

The next edge is (F, G) with a weight 8. Check whether including the edge (F, G) to the tree will create a cycle or not. It will create a cycle. So reject (F, G) from the tree. Fig. 5.4.22 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (F, G). We do not include this edge to the tree.

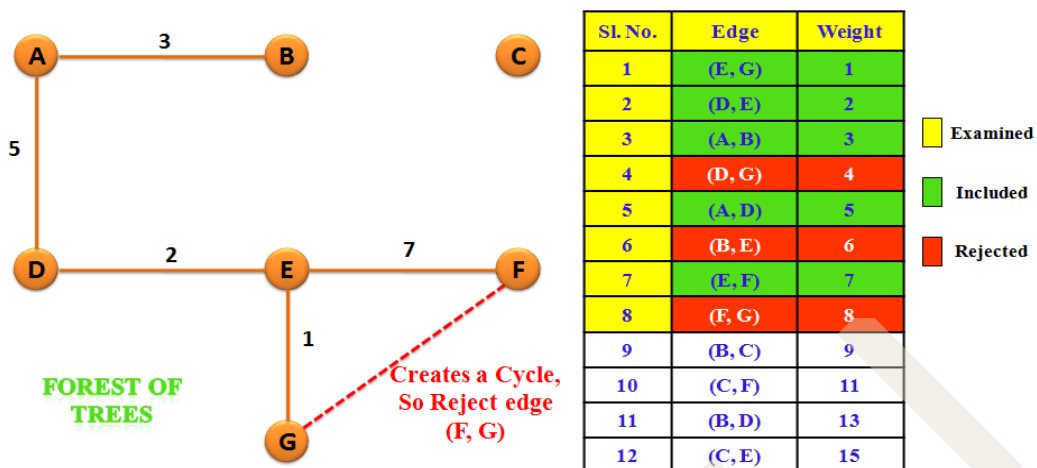


Fig 5.4.22 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (9).

Step 10:

The next edge is (B, C) with a weight 9. Check whether including the edge (B, C) to the tree will create a cycle or not. It will not create a cycle. So include (B, C) to the tree. Fig. 5.4.23 shows the resultant forest of trees and the priority queue.

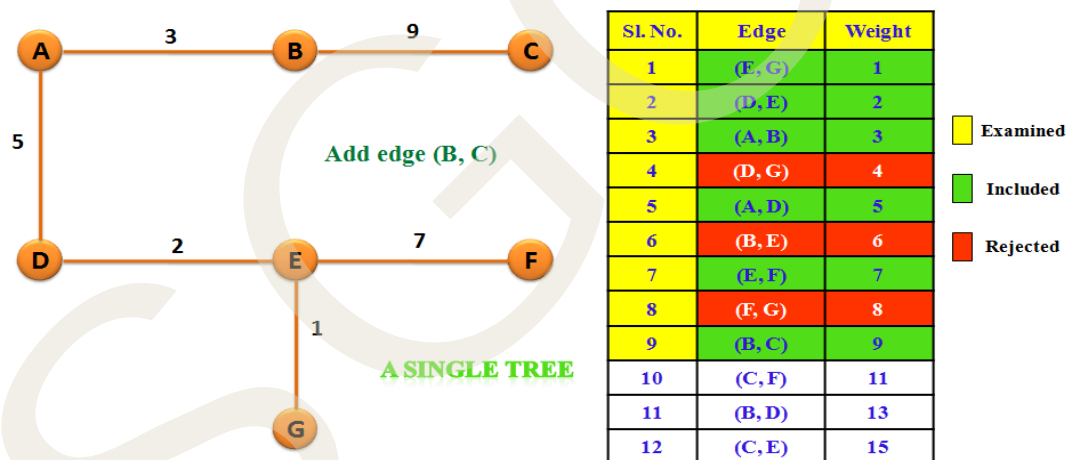


Fig 5.4.23 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (10)

Step 11:

The next edge is (C, F) with a weight 11. Check whether including the edge (C, F) to the tree will create a cycle or not. It will create a cycle. So reject (C, F) from the tree. Fig. 5.4.24 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (C, F). We do not include this edge to the tree.

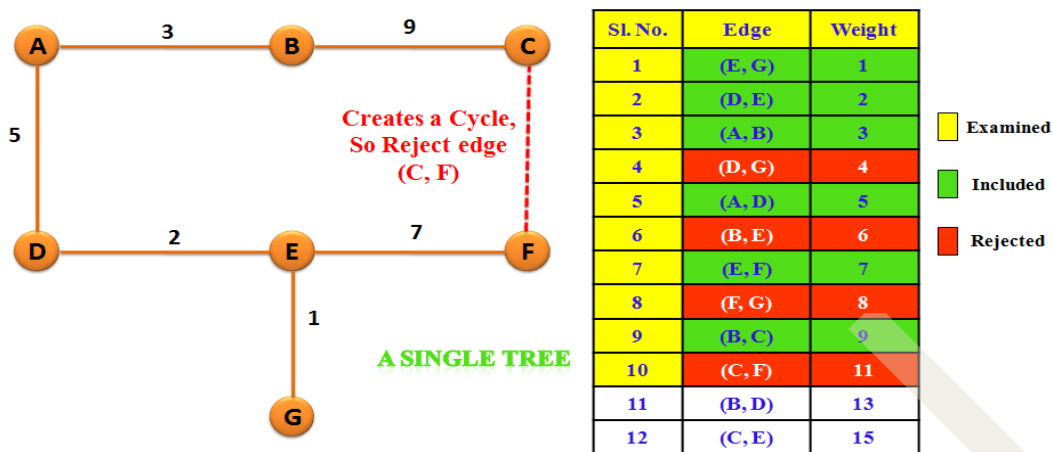


Fig 5.4.24 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (11)

Step 12:

The next edge is (B, D) with a weight 13. Check whether including the edge (B, D) to the tree will create a cycle or not. It will create a cycle. So reject (B, D) from the tree. Fig. 25 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (B, D). We do not include this edge to the tree.

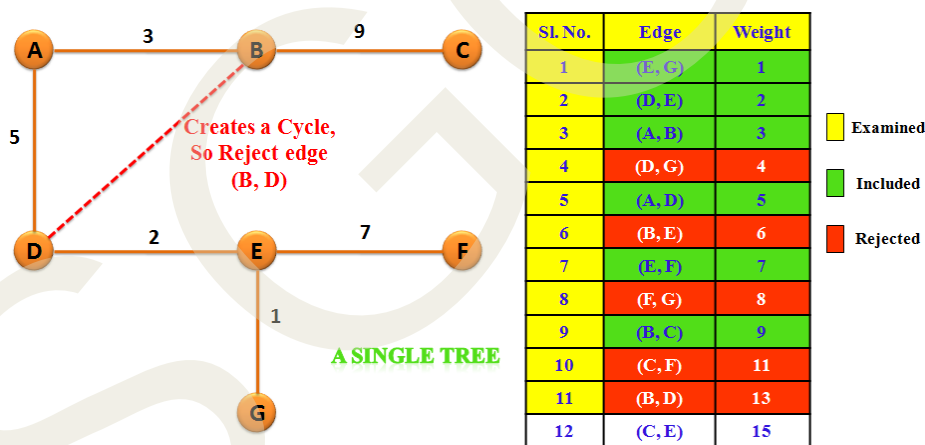


Fig 5.4.25 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (12).

Step 13:

The next edge is (C, E) with a weight 15. Check whether including the edge (C, E) to the tree will create a cycle or not. It will create a cycle. So reject the edge (C, E) from the tree. Fig. 5.4.26 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (C, E). We do not include this edge to the tree.

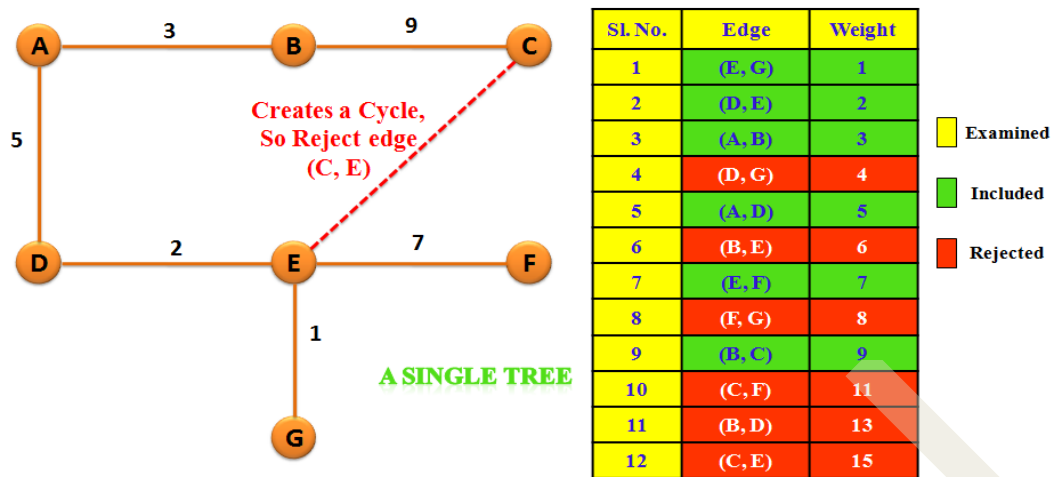


Fig 5.4.26 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (13)

And the final spanning tree is shown in Fig. 5.4.27. It is the minimum cost spanning tree of the given graph. The minimum cost is the sum of weights of all the edges present in the spanning tree. Here the sum is 27. So the minimum cost is 27.

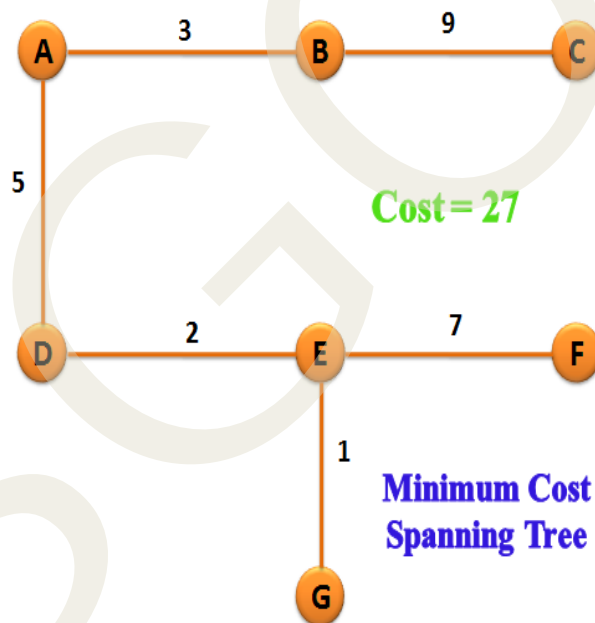


Fig 5.4.27 Kruskal's Algorithm – Final Minimum Cost Spanning Tree

5.4.4.1 Application of Kruskal's Algorithm

- ◆ Image Compression and Pattern Recognition: Used in building structures like region adjacency graphs to reduce data in images.
- ◆ Minimum Pipe Connection: Used in civil engineering for laying pipes to connect multiple locations at the lowest total cost.

- ◆ Telecommunication Systems: Useful in planning fiber optic networks where the goal is to minimize the total length of cables.⁴

5.4.5 Comparison between Prim's and Kruskal's Algorithms

Table 5.4.1 Comparison between Prim's and Kruskal's Algorithms

Aspect	Prim's Algorithm	Kruskal's Algorithm
1. Approach	Starts from any one vertex and expands the MST by adding the nearest connected vertex.	Starts with all edges sorted by weight and adds the smallest edge that connects two separate trees.
2. Edge Selection	Always selects the minimum-weight edge from the visited to unvisited vertices.	Always selects the globally smallest edge, regardless of its connection to previously added edges.
3. Cycle Detection	No explicit cycle detection is needed, as it always connects to a new vertex.	Requires explicit cycle detection using Union-Find (Disjoint Set) to avoid cycles.
4. Graph Suitability	Performs better on dense graphs, where many edges exist between vertices.	Works better on sparse graphs, where fewer edges mean quicker sorting and merging.
5. Implementation Complexity	Slightly more complex due to use of priority queues and maintaining visited status	Easier to implement once union-find is understood; mainly involves edge sorting and simple checks.

Recap

- ◆ A spanning tree is a subgraph that connects all vertices of a connected graph with exactly $n-1$ edges and no cycles.
- ◆ Every connected and undirected graph has at least one spanning tree.
- ◆ A disconnected graph cannot have a spanning tree.
- ◆ A complete graph with n vertices can have n^{n-2} spanning trees.
- ◆ Removing an edge from a spanning tree disconnects the graph.
- ◆ Adding an edge to a spanning tree creates a cycle.
- ◆ Spanning trees are used in network planning, clustering, and routing protocols.
- ◆ A minimum spanning tree (MST) is a spanning tree with the minimum possible total edge weight.

- ◆ MST connects all vertices while minimizing the total connection cost.
- ◆ Multiple MSTs can exist if the graph has edges with equal weights.
- ◆ MST is useful in telecom network design, map navigation, and utility grids.
- ◆ Prim's algorithm builds the MST by growing it from an arbitrary node, always selecting the nearest connected node.
- ◆ Prim's algorithm uses a priority queue and does not require explicit cycle checking.
- ◆ It is efficient for dense graphs with many edges.
- ◆ Kruskal's algorithm builds the MST by sorting all edges by weight and adding them one by one if they connect separate trees.
- ◆ Kruskal's algorithm uses the Union-Find data structure to avoid cycles.
- ◆ It is efficient for sparse graphs with fewer edges.
- ◆ Prim's algorithm selects the smallest edge connected to the growing MST.
- ◆ Kruskal's algorithm selects the smallest edge in the entire graph regardless of the current MST.
- ◆ Prim's algorithm is slightly more complex due to priority queue handling.
- ◆ Kruskal's algorithm is simpler if Union-Find is understood.
- ◆ Prim's algorithm is used in cable TV layout, road construction, and data clustering.
- ◆ Kruskal's algorithm is used in image processing, pipe connection planning, and fiber optic network design.

Objective Type Questions

1. What is a subgraph that includes all vertices and has no cycles?
2. Which algorithm starts from an arbitrary node and grows the MST?
3. What do we call a spanning tree with the smallest possible edge weights?
4. State the number of edges in a spanning tree of a graph with n nodes.
5. Which algorithm uses Union-Find to detect cycles?
6. What kind of graph does not have any spanning tree?

7. Which graph type has exactly one spanning tree?
8. Name a real-world application of MST.
9. State one data structure used in Kruskal's algorithm.
10. Which type of graph is best suited for Prim's algorithm?
11. What happens if we add one edge to a spanning tree?
12. What is the time complexity of Kruskal's algorithm with Union-Find?
13. What is the first criterion for selecting an edge in Prim's algorithm?
14. State the maximum number of spanning trees in a complete graph with n nodes.

Answers to Objective Type Questions

1. Spanning Tree
2. Prim's
3. Minimum Spanning Tree
4. $n-1$
5. Kruskal's
6. Disconnected
7. Tree
8. Networking
9. Disjoint Set
10. Dense
11. Cycle
12. $O(E \log E)$
13. Minimum Weight
14. n^{n-2}

Assignments

1. Explain the concept of a spanning tree with a real-time example.
2. Describe the properties of a spanning tree.
3. What is a minimum spanning tree? Explain with a suitable example.
4. Write the algorithm of Prim's algorithm and explain it step by step with an example graph.
5. Discuss the working of Kruskal's algorithm with a suitable example.
6. Compare Prim's and Kruskal's algorithms based on at least five aspects.
7. Describe the applications of minimum spanning trees in various fields.
8. How does Kruskal's algorithm ensure that cycles are not formed in the MST?

Reference

1. Srivastava, S. K., & Srivastava, D. (2023). *Data structures through C in depth*. BPB Publications..
2. Thareja, R. (2022). *Data structures using C* (2nd ed.). Oxford University Press..
3. E. Balagurusamy. (2020). *Fundamentals of Data Structures Using C*. McGraw-Hill Education.
4. Lipschutz, S. (2014). *Data Structures (Schaum's Outlines Series)*. McGraw-Hill Education.
5. Weiss, M. A. (2021). *Data structures and algorithm analysis in C* (2nd ed.). Pearson Education.

Suggested Reading

1. Yashavant Kanetkar. (2022). *Data Structures Through C*. BPB Publications.
2. A. M. Tenenbaum, Y. Langsam & M. J. Augenstein. (2022). *Data Structures Using C*. Pearson Education India.
3. Kruse, R. L., Leung, B. P., & Tondo, C. L. (1999). *Data structures and program design in C* (2nd ed.). Pearson Education.



Advanced Data Structures

Unit 1

Introduction to Advanced Data Structures

Learning Outcomes

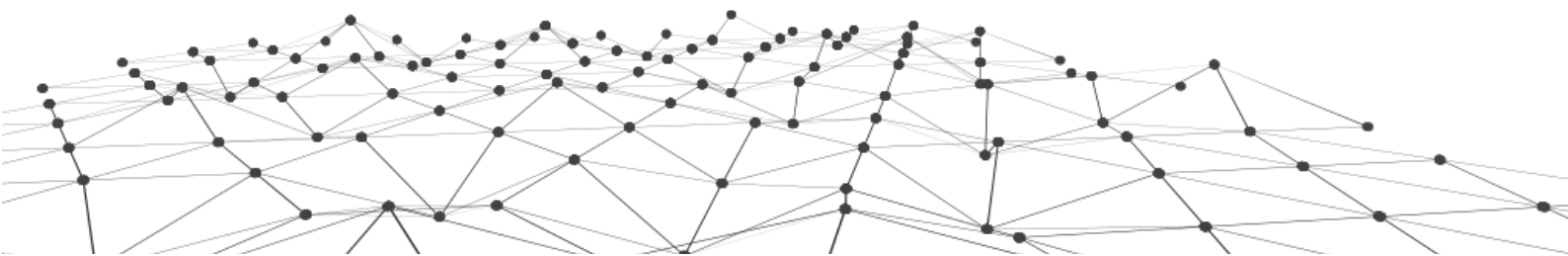
After the successful completion of the course, the learner will be able to:

- ◆ familiarise the need for advanced data structures in solving real-world problems.
- ◆ explain the properties and structure of Red-Black Trees.
- ◆ describe how Red-Black Trees maintain balance using rotations and recoloring.
- ◆ compare basic and advanced data structures using examples and time complexities.
- ◆ identify real-world applications of Red-Black Trees and other advanced structures.

Prerequisites

To effectively design and implement efficient software systems, it is essential to understand how data can be organized, accessed, and manipulated in the most optimal way. Basic data structures like arrays and linked lists often fall short when handling complex, large-scale, or performance-critical applications. This is where advanced data structures come in; they provide the foundation for faster algorithms and scalable systems. Studying them enables learners and developers to choose or design the right structure for specific needs, improving speed, memory usage, and maintainability.

In real life, these structures are at work behind the scenes in systems we use daily. For example, Red-Black Trees are used in memory management within operating systems, where quick allocation and deallocation of memory blocks are crucial. They are also widely used in databases and file systems to maintain ordered records with fast access times. Similarly, Tries help power predictive text features in search engines, and Heaps are used in job scheduling systems to manage task priorities. By mastering these structures, one can build software that is not just functional but also highly efficient and responsive.



Key words

Red-Black Tree, Self-balancing Tree, Tree Rotation, Logarithmic Time, Advanced Data Structures, Priority Queue

Discussion

In computer science, a data structure is a specialized format for organizing and storing data. It determines how efficiently various operations such as accessing, inserting, and deleting data can be performed. While basic data structures like arrays, stacks, and queues are suitable for simpler problems, they fall short in scenarios involving large datasets, frequent updates, or complex relationships between elements.

To meet these advanced requirements, computer scientists and engineers use **advanced data structures**. These include self-balancing trees (e.g., Red-Black Trees, AVL Trees), graphs, Tries, heaps, and hash tables. These structures are designed to maintain efficiency even under heavy or dynamic workloads, making them essential for modern software development.

6.1.1 significance in solving complex problems

Advanced data structures play a vital role in the efficient design and implementation of algorithms required to solve real-world, performance-critical problems. As modern computing applications continue to scale in terms of data volume and complexity, basic data structures such as arrays and linked lists become inadequate due to their inherent limitations in time and space efficiency. Advanced data structures such as Tries, Heaps, B-Trees, Graphs, and Balanced Trees offer specialized capabilities that allow for faster data retrieval, dynamic memory optimization, and real-time processing, which are essential for meeting the stringent demands of current technologies.

One prominent example of their significance is observed in the implementation of **autocomplete systems**, such as those used by search engines like Google. When a user types a few characters, the system must quickly return a list of matching suggestions from possibly millions of entries. A basic approach that sequentially checks each word in a list would be computationally expensive and slow. However, using a **Trie (prefix tree)** structure, where each node represents a character and paths from the root represent words, enables prefix lookups in linear time relative to the prefix length ($O(L)$). This dramatically improves the system's responsiveness and scalability.

Another critical application is found in **operating system scheduling**, where multiple processes must be managed efficiently based on priority. A naive list-based implementation would either require constant sorting or scanning, leading to performance degradation. In contrast, a **priority queue implemented with a Heap** (Max-Heap or Min-Heap) provides efficient insertion and deletion operations in logarithmic time ($O(\log n)$). This ensures that the highest-priority tasks are always selected for execution with minimal delay, making real-time task management feasible.

In the context of **database indexing**, advanced tree structures like **B-Trees and B+ Trees** are employed to maintain sorted data for rapid search, insertion, and deletion. These structures are inherently balanced and allow for logarithmic time operations, which is essential for maintaining performance in large-scale databases. Likewise, **graph-based structures** combined with algorithms like Dijkstra's are crucial in applications such as navigation and route planning, where optimal paths must be determined over millions of possible locations and roads. The use of a priority queue alongside adjacency lists enables efficient shortest-path computation with predictable performance.

Advanced data structures are indispensable tools for solving complex computational problems that demand high efficiency, scalability, and real-time responsiveness. By offering optimized mechanisms for storing, organizing, and accessing data, they enable the development of robust and intelligent systems across domains such as search engines, operating systems, databases, and navigation systems. Their strategic use ensures that even the most complex problems can be tackled within acceptable performance constraints, making them a cornerstone of modern computer science and software engineering.

6.1.2 Comparison between basic and advanced data structures

Data structures are foundational components of computer science, enabling efficient storage, access, and manipulation of data. They are broadly categorized into **basic data structures** and **advanced data structures**, each serving distinct roles depending on the complexity and requirements of the problem. **Basic data structures**, such as arrays, stacks, queues, and singly/doubly linked lists, are simpler in design and are primarily used for linear data organization and sequential operations. These structures are suitable for small-scale, less dynamic problems where operations are predictable and performance requirements are minimal. For instance, an **array** can be used to store a fixed-size list of student names, and a **stack** is well-suited for undo-redo functionality in text editors.

In contrast, **advanced data structures** are more sophisticated and are designed to support complex operations with improved time and space efficiency. They are typically employed in high-performance applications that require fast searching, frequent dynamic updates, or hierarchical and non-linear data relationships. Examples include **trees (e.g., AVL trees, Red-Black trees, B-Trees)**, **graphs**, **Tries**, **heaps**, and **hash tables**. For example, a **Red-Black Tree** is a self-balancing binary search tree used in system-level applications like memory allocation in operating systems and in the implementation of TreeMap in Java. A **Graph** is essential for modeling relationships in social networks or representing routes in GPS systems.

Suppose we are designing an online ticket booking system. Using a **queue** (basic structure), we can handle user requests in a first-come-first-serve manner. However, if we need to prioritize premium users, a **priority queue** (an advanced structure implemented using a heap) is more appropriate. Similarly, if we need to store and quickly retrieve booking records based on multiple conditions like date, seat class, and user ID, a **B-Tree** provides efficient logarithmic search performance even when dealing with millions of records.

The following table highlights the differences:

Table 6.1.1 Comparison Table of Basic vs. Advanced Data Structures

Criteria	Basic Data Structures	Advanced Data Structures
Examples	Array, Stack, Queue, Linked List	Trie, Red-Black Tree, Heap, Graph
Structural Complexity	Linear	Hierarchical or Networked
Balancing Mechanism	Not available	Self-balancing (e.g., AVL, RBT)
Time Complexity (Search)	$O(n)$ or $O(\log n)$	$O(\log n)$, $O(1)$ in some cases
Real-World Applicability	Educational/Small Projects	Large-scale systems, databases, OS
Scalability	Limited	High scalability for dynamic environments

6.1.3 Importance of Advanced Data Structures

In the evolving field of computer science, **Advanced Data Structures** play a vital role in solving large-scale, performance-intensive, and real-world problems. Unlike basic structures such as arrays and stacks, which are limited in flexibility and scalability, advanced structures like **Trees, Heaps, Tries, Graphs, Hash Tables, and Disjoint Sets** offer powerful solutions that are both **time-efficient** and **space-efficient**. As modern applications demand **real-time processing, instantaneous search, dynamic memory management, and intelligent data modeling**, advanced data structures become indispensable.

One key advantage of advanced data structures is **performance optimization**. For instance, searching in an unsorted list takes $O(n)$ time, but searching in a self-balancing binary search tree like a **Red-Black Tree** or **AVL Tree** reduces it to $O(\log n)$. This becomes crucial in systems handling large datasets, like databases and search engines. Similarly, **Tries** enable fast prefix matching in $O(L)$ time (L = length of word), which is the backbone of real-time autocomplete systems like Google Search.

In addition, **efficient memory usage and priority handling** are vital in operating systems and embedded systems. For example, **Min-Heaps** are used for CPU task scheduling, where processes with the highest priority (lowest value) are executed first. **Hash tables** are used for constant-time lookups in compilers, caching, and key-value stores.

Learning advanced data structures helps:

- ◆ **Optimize algorithms:** Improve performance for time-sensitive applications.

- ◆ **Solve competitive coding challenges:** Frequently asked in coding rounds.
- ◆ **Understand systems:** Used in compilers, databases, networking, and operating systems.
- ◆ **Develop scalable software:** Handle millions of records efficiently.

For example, **database indexes** are built using **B+ Trees**, and **operating systems** use **Red-Black Trees** to manage memory or I/O scheduling.

Table 6.1.2 Applications and Importance of Common Advanced Data Structures

Advanced Data Structure	Application Area	Why It's Important
Trie	Autocomplete, Spell Checker	$O(L)$ prefix search; ideal for dictionary systems
Heap (Min/Max Heap)	CPU Scheduling, Dijkstra's Algorithm	Efficient priority queue operations
Red-Black Tree	Databases, OS File Indexing	Balanced tree guarantees $O(\log n)$ operations
Graph	GPS Routing, Social Networks, Web Crawling	Represents and traverses complex relationships
Hash Table	Caches, Symbol Tables in Compilers	Fast lookup, insertion, and deletion ($O(1)$ avg.)
Disjoint Set (Union-Find)	Kruskal's MST, Network Connectivity	Efficient grouping and connectivity checking

Real-Life Use Cases

1. Google Autocomplete System

When a user types a query like “mach...”, the system instantly suggests phrases like “machine learning” or “machine gun”. This is achieved using a **Trie** that allows prefix-based searching in linear time with respect to the prefix length.

2. CPU Scheduling in Operating Systems

The Linux kernel uses a **priority queue implemented with heaps** to determine which task gets CPU time next, ensuring high-priority tasks are not starved.

3. Social Network Recommendations

Platforms like Facebook use **graph algorithms** to suggest friends by traversing the social graph to find mutual connections using **BFS (Breadth-First Search)**.

6.1.4 Red-Black Trees

A **Red-Black Tree (RBT)** is a **self-balancing Binary Search Tree (BST)** that maintains a balanced structure by assigning a color (red or black) to each node and applying rules during insertion and deletion.

The tree avoids skewed structures that degrade performance, ensuring that all operations such as **search**, **insert**, and **delete** take $O(\log n)$ time.

Red-Black Tree Node Structure

Each node in a Red-Black Tree typically contains:

```
struct Node {  
    int key;  
    char color;        // 'R' for Red, 'B' for Black  
    Node *left, *right, *parent;  
};
```

6.1.4.1 Properties of Red-Black Trees

A tree qualifies as a Red-Black Tree if it satisfies the following rules:

1. **Node Color:** Each node is colored either red or black.
2. **Root Node:** The root node must always be black.
3. **Leaf Nodes:** All leaf nodes (NIL or NULL pointers) are considered black.
4. **No Double Red:** A red node cannot have a red child. No two consecutive red nodes are allowed.
5. **Black-Height Consistency:** Every path from a node to its leaf descendants must have the same number of black nodes.

These rules ensure that the longest path is no more than twice the length of the shortest path → guaranteeing logarithmic height.

A Red-black tree with n number of internal nodes has a height of $2\log(n+1)$.

6.1.4.2 Balancing in Red-Black Trees

Importance of balancing

Balancing is a key feature of Red-Black Trees. A balanced tree ensures that the height of the tree stays logarithmic with respect to the number of nodes. This is important because the time complexity of searching, inserting, and deleting elements in a tree is directly related to its height. Without balancing, a binary search tree (BST) could become skewed (like a linked list), degrading the time complexity to $O(n)$. Red-Black Trees solve this issue by automatically rebalancing themselves during insertions and deletions.

How Balancing Is Maintained

Red-Black Trees maintain balance through a combination of coloring rules and rotations. Each node in the tree is colored either red or black, and the tree follows a strict set of properties (rules mentioned above) to ensure it stays balanced. These rules ensure that no path in the tree is more than twice as long as any other, maintaining approximate balance.

Rotations for Balancing

To restore these properties after an insertion or deletion, the Red-Black Tree uses **tree rotations**. These are local operations that restructure part of the tree without breaking the binary search tree property.

Table 6.1.3 Types of Rotations

Rotation Type	Description
Left Rotation	Performed when a right child causes imbalance. Moves the right child up and the current node down to the left.
Right Rotation	Performed when a left child causes imbalance. Moves the left child up and the current node down to the right.

In some cases, a combination of left and right rotations (called Left-Right or Right-Left rotations) is used.

Balancing After Insertion

1. Insert as in a normal BST (new node is red).
2. Fix violations of Red-Black properties using:
 - ◆ Recoloring (change node colors).
 - ◆ Rotations (Left or Right rotation).

Cases in Insertion Fixing:

- ◆ Case 1: Parent is black → No violation.
- ◆ Case 2: Parent is red & Uncle is red → Recolor parent and uncle to black, grandparent to red.
- ◆ Case 3: Parent is red & Uncle is black → Perform rotation and recoloring.

Uncle in a Red-Black Tree

When we talk about uncle in the context of insertion or balancing in Red-Black Trees:

- ◆ A node's uncle is the sibling of its parent.
- ◆ In other words, if a node's parent has a brother, that brother is the uncle of the node.

Suppose we insert a new red node into the tree. If this causes a violation of the red-black properties (e.g., two red nodes in a row), the tree performs the following:

- ◆ Recoloring: Adjusts the color of nodes to restore the black-height.
- ◆ Rotation: Reorganizes the structure to fix any violations caused by the insertion.

Algorithm (High-Level Steps for Insertion Balancing)

Insert the new node as in a regular BST (color it red)

While the parent node is red:

Case 1: Uncle node is red

Recolor parent and uncle to black

Recolor grandparent to red

Move up the tree

Case 2: Uncle node is black and the new node forms a triangle

Rotate to make a line (left or right rotation)

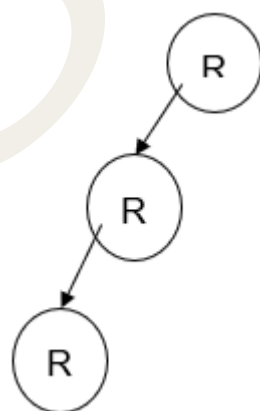
Case 3: Uncle node is black and the new node forms a line

Perform rotation and recolor to fix the violation

Ensure root is black at the end

Example

Before Insertion (violation):



← Two red nodes in a row

Fig. 6.1.1 Before insertion

After Rotation and Recoloring:

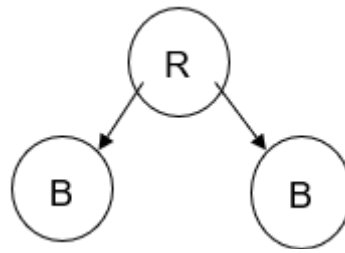


Fig. 6.1.2 After rotation and recoloring

This simple rotation and recoloring operation restores the red-black properties and keeps the tree balanced.

Example :Insert the elements 8, 20, 11, 14, 9, 4, 12 in a red black tree.

Insert 8

If the tree is an empty tree, then make node 8 as the root and color it black.



Fig 6.1.3 Insert 8

All four properties of the red black tree met.

Insert 20

$20 > 8$, so node 20 will be the right child of node 8 and its color is red.

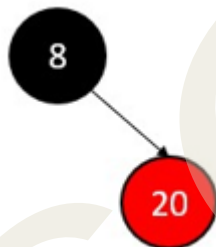


Fig 6.1.4 Insert 20

All four properties of the red black tree met.

Insert 11

$11 < 20$, so node 11 will be the left child of node 20 and its color is red.

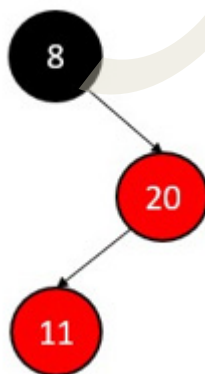


Fig 6.1.5 Insert 11

Every red node has black children, but here 20 is a red parent having left child red 11.

But recoloring is not possible, as the property of black height will be violated. All we have to do is Right-left rotation.

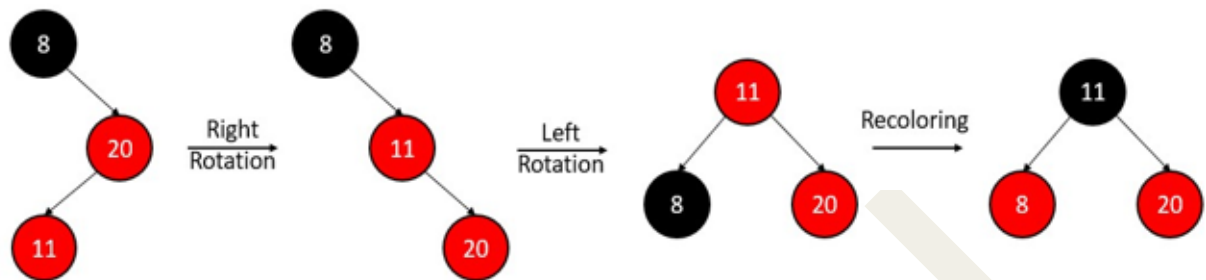


Fig 6.1.6 Right-left rotation

All four properties of the red black tree met.

Insert 14

$14 < 20$, so node 14 will be the left child of node 8 and its color is red.

The property “every red node has black children” is violated. So, recolor the node 8 and node 20 black.

All four properties of the red black tree met.

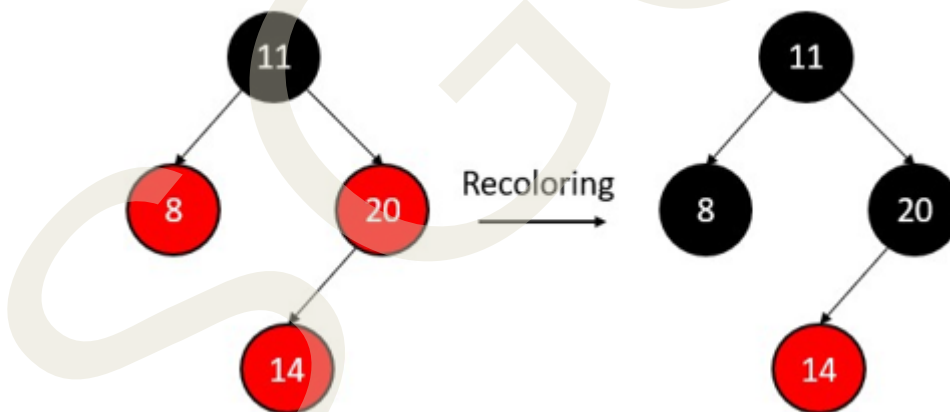


Fig 6.1.7 Recoloring

Insert 9

$9 > 8$, so node 9 will be the right child of node 8 and its colour is red.

All four properties of the red black tree met.

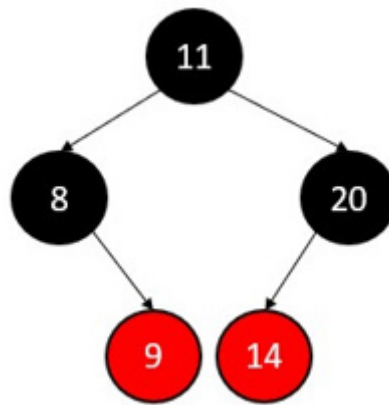


Fig 6.1.8 Insert 9

Insert 4

All four properties of the red black tree met.

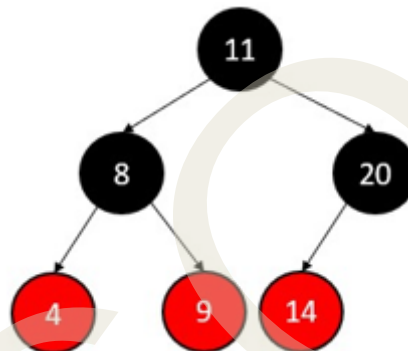


Fig 6.1.9 Insert 4

Insert 12

All four properties of the red black tree met.

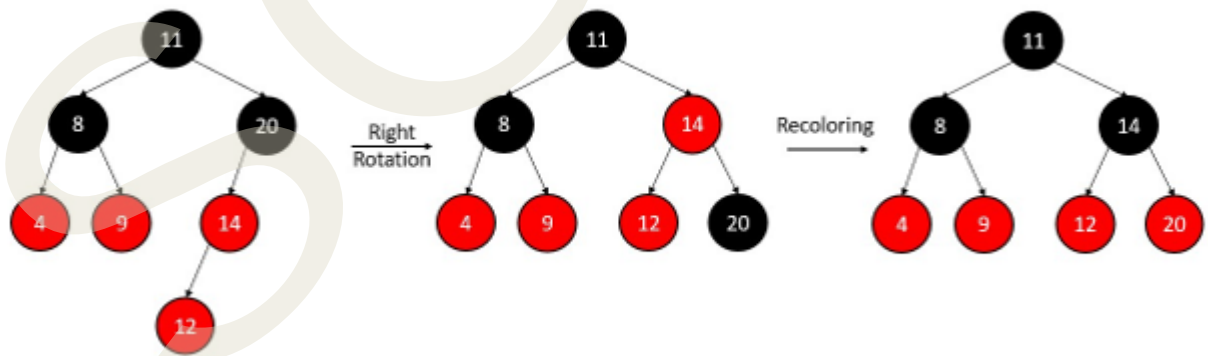


Fig 6.1.10 Insert 12

Balancing After Deletion

If you delete a node in the red black tree, you may violate the previously specified property. We recolor the tree nodes to match all properties, and if that doesn't work, we rotate them.

To delete a node n , **identify its predecessor in the left subtree** and replace it with the deleted node.

When a node is deleted from a Red-Black Tree, particularly a **black node**, it can disturb the critical property of **equal black-height** across all paths from root to leaf. This imbalance introduces what is known as a “**double black**” node, a temporary placeholder indicating that one extra black level is needed to maintain balance. The rebalancing process involves checking the **sibling node of the double black** and applying different strategies based on the sibling’s color and the color of the sibling’s children. These strategies include **recoloring**, **rotations**, and sometimes **propagating the double black issue upwards** to the parent.

The main goal is to eliminate the double black without violating any of the Red-Black Tree properties, especially the rules that:

The number of black nodes must be the same on all paths from root to leaf.

Red nodes cannot have red children.

Table 6.1.4 Balancing Cases After Deletion

Case	Condition	Action
Case 1	Sibling is red	Rotate the parent in the opposite direction. Swap colors of parent and sibling. Proceed to the next case.
Case 2	Sibling is black , and both of its children are black	Recolor the sibling to red. Remove one black from the double black node and push it up.
Case 3	Sibling is black , and near child (closer to double black) is red	Rotate siblings toward double black. Swap sibling’s and near child’s colors. Move to Case 4.
Case 4	Sibling is black , and far child (opposite side) is red	Rotate parent around double black. Swap parent’s and sibling’s colors. Recolor far child black. Balancing completely.

Example for deletion
Input is

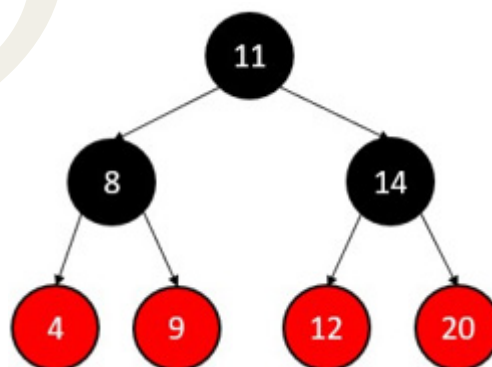


Fig 6.1.11 Input tree

Delete 12

Node 12 has neither a predecessor nor a successor because it has no children. So, delete this node and verify all the properties are met after deletion.

All four properties of the red black tree met after deletion.

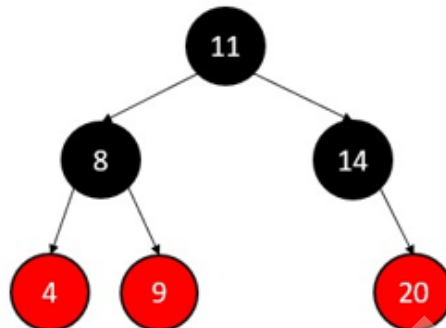


Fig 6.1.12 Delete 12

Delete 14

Node 14 has the right child. It has no predecessor, but it has a successor, 20. Its parent node 11 will become the parent of its child node 20. Because 20 is red, it violates the same black height as every path. After recoloring, node 20 will be a black node, and all four conditions of the red-black tree will be met.

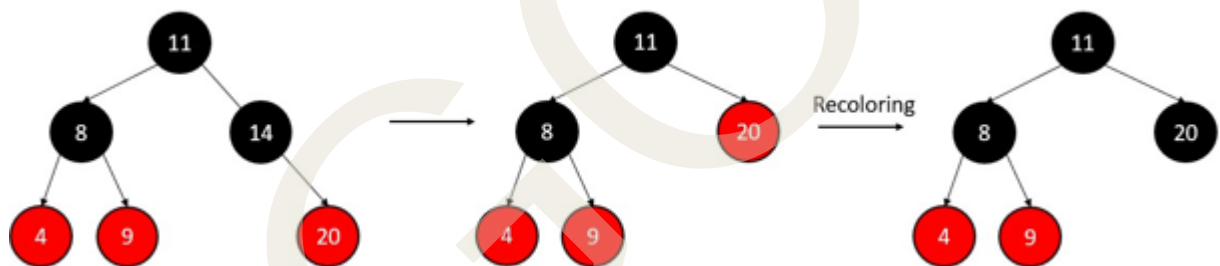


Fig 6.1.13 Delete 14

Delete 11

Node 11 has both children. So, find the predecessor in the left subtree of the node.

The predecessor is 9. So, replace the node 11 by node 9 and color it black, because 11 is the root node of the tree.

After deletion, all four properties of the red-black tree are met.

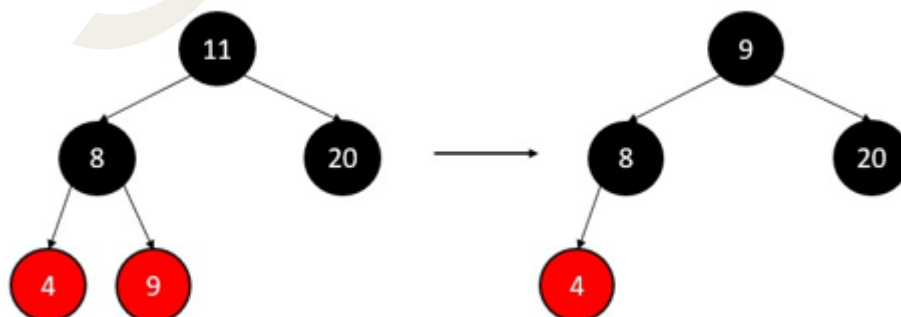


Fig 6.1.14 Delete 11

6.1.4.3 Advantages of Red-Black Trees

1. **Self-balancing:** Maintains balanced tree structure automatically to ensure optimal performance.
2. **Efficient operations:** Search, insertion, and deletion all operate in $O(\log n)$ time in the worst case.
3. **Easier to implement** than other balanced trees like AVL trees.
4. **Widely used** in real-world systems such as TreeMap (Java) and std::map (C++).
5. **Consistent performance** even during frequent operations.

6.1.4.4 Disadvantages of Red-Black Trees

1. **More complex** than regular Binary Search Trees due to color rules and rotations.
2. **Insertion/deletion overhead:** Additional operations required to maintain red-black properties.
3. **Not optimal for all scenarios:** May not perform best when frequent insertions and deletions are required compared to AVL or simpler structures.

6.1.4.5 Applications of Red-Black Trees

1. **Maps and Sets:** Used in many programming languages (e.g., Java TreeMap, C++ std::map) for storing sorted key-value pairs.
2. **Priority Queues:** Helps in implementing priority queues with efficient updates and ordering.
3. **File Systems:** Used to manage hierarchical structures of files and directories (e.g., Linux file systems).
4. **In-Memory Databases:** Stores and retrieves sorted data quickly.
5. **Game Development:** Assists in collision detection, pathfinding, and efficient object updates.

Recap

- ◆ Advanced data structures are specialized tools used to efficiently solve complex and large-scale computational problems.
- ◆ Red-Black Trees are self-balancing binary search trees that maintain balance through color and rotation rules.
- ◆ The height of a Red-Black Tree is always $O(\log n)$, ensuring efficient search, insertion, and deletion.
- ◆ Every node in a Red-Black Tree is either red or black, with specific rules ensuring balance and structure.
- ◆ No two consecutive red nodes are allowed on any path in a Red-Black Tree.
- ◆ The root node of a Red-Black Tree is always black, and all paths from root to leaf have the same number of black nodes.
- ◆ Rebalancing after insertion or deletion involves rotations (left or right) and node recoloring.
- ◆ Red-Black Trees provide better performance in insertions and deletions compared to AVL Trees due to their less strict balancing.
- ◆ Common operations on Red-Black Trees (search, insert, delete), all run in worst-case $O(\log n)$ time.
- ◆ Red-Black Trees are widely used in implementing associative containers like maps and sets.
- ◆ Tries are used in autocomplete systems for fast prefix matching in $O(L)$ time, where L is the length of the query.
- ◆ Heaps (like min-heap or max-heap) are used in operating systems and algorithms for priority queue management.
- ◆ Red-Black Trees are used in file systems, in-memory databases, and real-time applications where balanced performance is critical.
- ◆ The balance and performance of advanced data structures are key in building scalable and responsive applications.
- ◆ Compared to basic structures, advanced data structures offer better time and space complexity for dynamic and large data.

Objective Type Questions

1. Which advanced tree structure is self-balancing and uses colors for balance?
2. What is the time complexity of searching in a Red-Black Tree in the worst case?
3. Which data structure is used in autocomplete systems for efficient prefix search?
4. What type of rotation is used in Red-Black Trees to maintain balance?
5. Which advanced data structure is commonly used to implement priority queues?
6. What is the color of the root node in a Red-Black Tree?
7. What ensures equal black nodes on all paths in a Red-Black Tree?
8. Which heap ensures the smallest element is always at the root?
9. Which tree is stricter in balancing than Red-Black Trees?
10. What type of tree is used in many file systems for directory management?

Answers to Objective Type Questions

1. Red-Black
2. Logarithmic
3. Trie
4. Left
5. Heap
6. Black
7. Black-height
8. Min-heap
9. AVL
10. Red-Black

Assignments

1. Define advanced data structures. Why are they preferred in modern computing applications?
2. Explain with an example how a Trie supports prefix search.
3. Describe the importance of balancing in Red-Black Trees.
4. Illustrate the insertion of nodes 8, 20, 11, and 14 in a Red-Black Tree and explain each step.
5. Compare and contrast Red-Black Trees and AVL Trees in terms of balancing and performance.
6. Discuss real-world applications where Red-Black Trees are used and justify their importance.
7. Explain the process of balancing a Red-Black Tree after deletion with a suitable case.
8. Create a table comparing time complexity of basic vs. advanced data structures.
9. What role does rotation play in Red-Black Trees? Describe left and right rotation scenarios.
10. Highlight the advantages and disadvantages of using Red-Black Trees in system-level programming.

Reference

1. Horowitz, E., Sahni, S., & Mehta, D. (2007). *Fundamentals of Data Structures in C++* (2nd ed.). Universities Press.
2. Kruse, R. L., Leung, B. P., & Tondo, C. L. (1997). *Data Structures and Program Design in C* (2nd ed.). Pearson.
3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
4. Drozdek, A. (2012). *Data Structures and Algorithms in C++* (4th ed.). Cengage Learning.
5. McConnell, J. J. (2007). *Analysis of Algorithms: An Active Learning Approach* (2nd ed.). Jones & Bartlett Learning.

Suggested Reading

1. Horowitz, E., Sahni, S., & Mehta, D. (2007). *Fundamentals of Data Structures in C++* (2nd ed.). Universities Press.
2. Kruse, R. L., Leung, B. P., & Tondo, C. L. (1997). *Data Structures and Program Design in C* (2nd ed.). Pearson.
3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
Drozdek, A. (2012). *Data Structures and Algorithms in C++* (4th ed.). Cengage Learning.
4. McConnell, J. J. (2007). *Analysis of Algorithms: An Active Learning Approach* (2nd ed.). Jones & Bartlett Learning.

Unit 2

Introduction to Hashing

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- ◆ define hashing and explain its purpose in data structures
- ◆ identify the components of a hash table
- ◆ recall the different collision resolution techniques used in hash tables
- ◆ familiarize the types of open addressing methods

Prerequisites

Imagine you have a box full of hundreds of student records and you are asked to find one student's details. If you check each record one by one, it will take a lot of time. Computers also face the same problem when they have to search through large amounts of data. Earlier, you learned basic data structures like arrays and linked lists. In arrays, data is stored at numbered positions (indexes), and in linked lists, elements are connected one after another. But when we search for an item in these structures, we may need to go through each element, which takes more time. To solve this problem, we use a method called hashing, which stores data in a special structure called a hash table. It uses a hash function to find the exact location of data directly, without checking every item. Understanding how data is stored in arrays and how linked lists work will help you easily understand how hashing makes searching, inserting, and deleting data much faster.

Key words

Collision, Chaining, Open Addressing, Linear Probing, Quadratic Probing, Double Hashing



Discussion

6.2.1 Introduction to Hashing

Hashing is a technique used to efficiently store and retrieve data in a data structure known as a hash table. It involves transforming input data (often called keys) into a fixed-size string of bytes, typically using a hash function. This transformation maps the input data to a unique index in a hash table, allowing for fast data access. Hashing is widely used in various applications, including databases, caches, and data structures such as hash maps or hash sets, to achieve constant-time complexity for operations like search, insert, and delete.

6.2.1.1 Hash Tables

A hash table is a unique type of data structure that allows fast storage and retrieval of information using a key. You can think of it like a large table where each entry has a label (the key) and some data (the value).

When you add data, you assign it a key, and a special mathematical formula called a hash function figures out the exact spot to store it as in Fig 6.2.1. Later, if you need to get that data back, you use the same key, and the hash table quickly locates it for you.

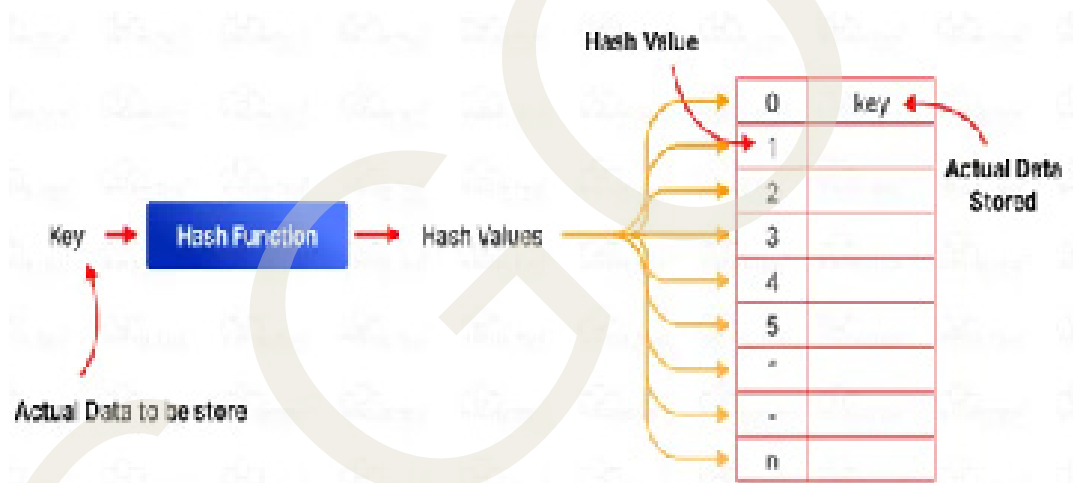


Fig 6.2.1 Hash Table

Hash tables play a key role in data structures and algorithms (DSA) because they allow much faster operations like searching, inserting, and deleting than structures such as lists or arrays. This speed makes them ideal for use in applications like dictionaries, caches, and databases, where fast data access is essential.

6.2.1.2 Hash Functions

A hash function is a mathematical formula that transforms a key into a specific index in a hash table. It guarantees that the same key will always generate the same index. For instance, the key “apple” might be turned into index 5, while “banana” could map to index 8. An effective hash function spreads keys evenly throughout the table to reduce the chances of collisions.

6.2.2 Collision Resolution Techniques

In hashing, a collision happens when two or more different keys are assigned the same index by the hash function in the hash table as in Fig 6.2.2. Since the hash table relies on these indexes for fast data storage and retrieval, collisions create issues because multiple items would compete for the same position.

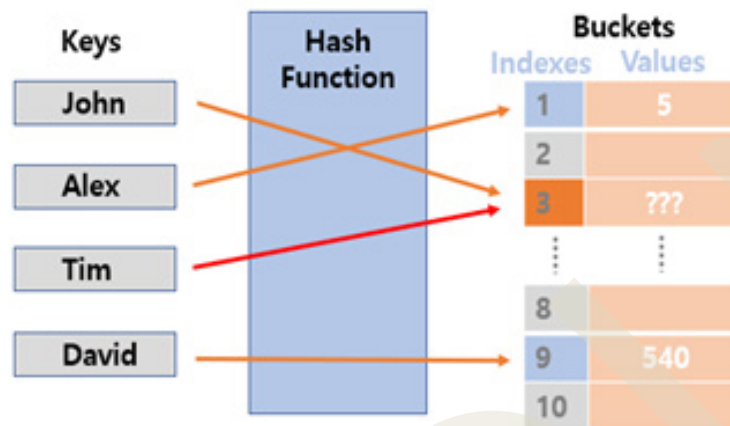


Fig 6.2.2 Collision

To manage collisions and maintain the efficiency of the hash table, several collision resolution methods are applied. These methods allow the hash table to store and access data quickly, even when different keys share the same index, ensuring no data is lost.

Common collision resolution methods include

- ◆ Chaining
- ◆ Open Addressing

6.2.2.1 Chaining

In a hash table, each index is connected to a linked list as in Fig 6.2.3. When multiple keys hash to the same index, their corresponding values are added to the linked list at that location. This approach ensures that, even when collisions occur, all the values can still be retrieved by going through the linked list at that index.

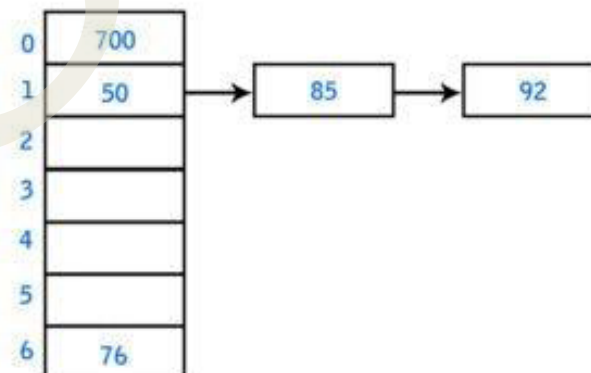


Fig 6.2.3 Chaining

When a single slot contains multiple elements such as {50, 85, 92}, a linked list is used to store the extra items {85, 92} at that index. Using the **chaining** method makes inserting or deleting elements in a hash table relatively easy and efficient. However, since chaining relies on linked lists, it also shares their advantages and limitations. In some cases, **dynamic arrays** can be used instead of linked lists for implementing chaining.

6.2.2.2 Open Addressing

In open addressing, entries are stored directly in the array rather than using linked lists. The hash value does not directly determine the final location of an item. Instead, the algorithm starts checking from the hashed index and uses a **probing sequence** to find the next available empty slot. This sequence determines how the search moves through the array, often with varying gaps between checks. This method is also known as **Closed hashing**, and it handles collisions using three main techniques.

- ◆ Linear Probing
- ◆ Quadratic Probing
- ◆ Double-Hashing

1. Linear Probing

Linear probing involves checking the hash table in a step-by-step manner starting from the hashed index as in Fig 6.2.4. If the desired spot is already taken, the next available slot is searched. In this method, the gap between each probe is usually constant, commonly set to 1.

Formula:

$$\text{index} = \text{key} \% \text{hashTableSize}$$

Linear Probing Example

Insert (76)	Insert (93)	Insert (40)	Insert (47)	Insert (10)	Insert (55)
$76\%7 = 6$	$93\%7 = 2$	$40\%7 = 5$	$47\%7 = 5$	$10\%7 = 3$	$55\%7 = 6$
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	6	6	6	6
76	93	40	47	10	55

Fig 6.2.4 Linear Probing

2. Quadratic Probing

The main difference between linear and quadratic probing lies in the spacing between the slots checked during a collision. In quadratic probing, if a collision occurs, you continue searching for an open slot, but the gap between each probe increases based on a polynomial pattern. This means the next index is calculated by adding squared values (or other polynomial increments) to the original hashed index as in Fig 6.2.5.

Formula:

$$\text{index} = \text{index} \% \text{hashTableSize}$$

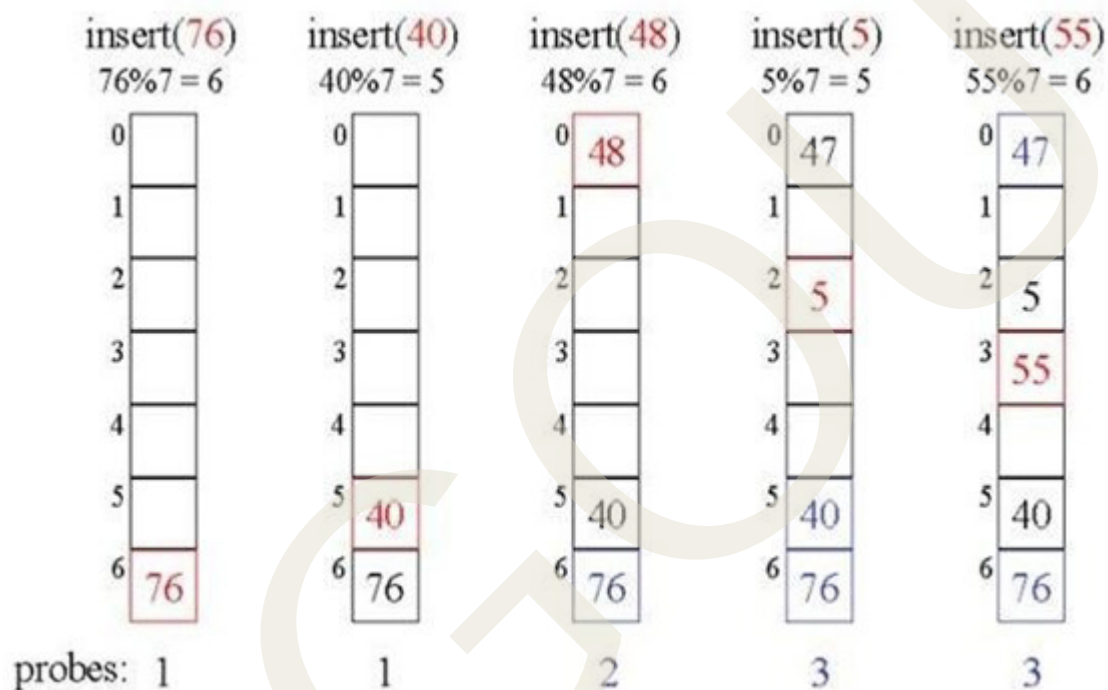


Fig 6.2.5 Quadratic Probing

3. Double-Hashing

In double hashing, the interval between probes is calculated using a second hash function. This method helps reduce clustering more effectively than other probing techniques. The steps taken during probing are determined by this additional hash function, making the process more efficient and evenly distributed.

Formula

$$(\text{first hash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{size of the table}$$

Recap

- ◆ Hashing is used to store and retrieve data efficiently using a hash table.
- ◆ Uses a hash function to map keys to fixed-size indexes.
- ◆ Provides fast operations (search, insert, delete) — often in constant time.

Hash Tables

- ◆ Data structure storing key–value pairs.
- ◆ Uses a hash function to find the index where data is stored.
- ◆ Enables quick access, widely used in dictionaries, caches, and databases.

Hash Functions

- ◆ Convert keys into specific indexes.
- ◆ Same key → same index consistently.
- ◆ Should distribute keys evenly to avoid collisions.

Collision Resolution Techniques

- ◆ Collision occurs when two keys map to the same index.
- ◆ Needs resolution to maintain efficiency and prevent data loss.
- ◆ Two main approaches: Chaining and Open Addressing.

Chaining

- ◆ Each index stores a linked list of items with the same hash index.
- ◆ Easy insertion and deletion.
- ◆ May use dynamic arrays instead of linked lists.

Open Addressing (Closed Hashing)

- ◆ All entries stored directly in the hash table array.
- ◆ On collision, uses probing to find next free slot.

Types of probing:

- ◆ **Linear Probing** – checks next slots one by one (gap = 1).
 - Formula: $\text{index} = \text{key} \% \text{hashTableSize}$
- ◆ **Quadratic Probing** – checks slots at increasing squared gaps.
 - Formula: $\text{index} = (\text{index} + i^2) \% \text{hashTableSize}$
- ◆ **Double Hashing** – uses a second hash function to determine step size.
 - Formula: $(\text{firstHash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{tableSize}$

Objective Type Questions

1. What data structure is primarily used to store key–value pairs using hashing?
2. What is the process of mapping keys to indexes called?
3. What is the function used to convert keys into table indexes called?
4. What term describes two different keys producing the same index?
5. Which technique connects a list of elements at the same index to handle collisions?
6. Which collision resolution technique stores all elements within the table array itself?
7. Which probing method checks the next immediate slot on collision?
8. Which probing method increases the gap between probes using a square pattern?
9. Which probing method uses a second hash function to find the gap between probes?
10. What type of hashing is Open Addressing also known as?

Answers to Objective Type Questions

1. HashTable
2. Hashing
3. HashFunction
4. Collision
5. Chaining
6. OpenAddressing
7. Linear
8. Quadratic
9. DoubleHashing
10. Closed

Assignments

1. Explain the working of a hash table with a neat diagram.
2. Describe the characteristics of a good hash function.
3. What are collisions in hashing?
4. Compare chaining and open addressing techniques for handling collisions.
5. Describe linear probing, quadratic probing, and double hashing.
6. Discuss how chaining handles collisions in a hash table.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
2. Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.
3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.
4. Lafore, R. (2002). *Data structures and algorithms in Java* (2nd ed.). Sams Publishing.

Suggested Reading

1. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
2. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.
3. Shaffer, C. A. (2013). *Data structures and algorithm analysis in Java* (3rd ed.). Dover Publications.
4. Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson.
5. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). *Fundamentals of data structures in C* (2nd ed.). Silicon Press.

Unit 3

Heap Data Structure

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- ◆ recall the definitions, properties, and types of heaps such as min-heap and max-heap.
- ◆ explain the structure and working of a binary heap with suitable examples and diagrams.
- ◆ differentiate between binary, Fibonacci, binomial, and pairing heaps based on their features and performance.
- ◆ evaluate the advantages and limitations of various heap representations in real-world applications such as priority scheduling and shortest path algorithms.

Prerequisites

Imagine a situation at a busy hospital emergency room. Patients arrive with different levels of urgency, some with minor injuries and others in critical condition. The staff cannot treat patients in the order they arrive; instead, they must quickly decide who needs attention first. This is a real-world example of prioritization. In the world of computing, similar decisions are made by operating systems when they handle multiple tasks or processes. High-priority tasks, like system alerts or real-time data processing, must be handled before less urgent ones. This need to manage tasks based on importance rather than arrival time is where the concept of heaps becomes essential. The heap data structure provides a reliable way to organize and retrieve elements based on priority, making it a fundamental tool for solving problems in scheduling, simulation, network routing, and many other areas of computer science.

To understand heaps effectively, learners should already be familiar with arrays, tree structures, recursion, and basic algorithmic analysis. Studying heaps helps learners see how data can be managed and processed efficiently, especially when speed and order of importance matter. It also builds a deeper understanding of how algorithms like Heapsort, Dijkstra's shortest path, and Prim's algorithm function. The study of heaps goes beyond theory; it teaches how to think logically and make optimal decisions. To develop interest and motivation for this topic, learners can relate heap operations to familiar situations, such as arranging exam priorities, selecting tasks in a to-do list,



or choosing which customer request to respond to first in a helpdesk system. Using hands-on coding activities, flow diagrams, and real-time examples in class can make the topic more engaging and help learners build both confidence and problem-solving skills.

Key words

Heaps, Binary Heap, Min-Heap, Max-Heap, Heapify, Binary Heap, Insertion, Deletion

Discussion

6.3.1 Introduction to Heap

A heap is a specialized tree-based data structure that satisfies the heap property and is primarily used to implement priority queues. It is typically implemented as a complete binary tree, which means that all levels are completely filled except possibly the last, which is filled from left to right. The structural property of the heap ensures that the tree remains balanced and allows for efficient performance of priority-based operations. The heap property defines the ordering of elements in a heap and ensures that the root always holds the most important value.

Heaps are an essential tool in situations where elements need to be managed according to priority rather than the order in which they arrive. This makes heaps a key building block in systems where timely execution and decision-making are critical. Examples include task scheduling in operating systems, data stream processing, event-driven simulations, and several fundamental graph algorithms.

A heap provides a concrete implementation of a priority queue, where each element is associated with a priority. The heap property ensures that the element with the highest or lowest priority (depending on the heap type) is always at the root of the structure. This guarantees that the most important element can be accessed quickly and efficiently.

The heap data structure is widely used in numerous real-world applications due to its ability to efficiently manage elements based on priority. One of its most prominent uses is in the implementation of priority queues, where tasks or elements are processed based on urgency rather than arrival order, such as in operating system schedulers and network routers. Heaps also form the backbone of the Heapsort algorithm, a comparison-based sorting technique that organizes elements by first converting the dataset into a heap and then repeatedly extracting the root to produce a sorted sequence in $O(n \log n)$ time. In graph algorithms, heaps are crucial; for instance, Dijkstra's algorithm employs a min-heap to continually select the vertex with the smallest tentative distance, enabling efficient computation of shortest paths. Similarly, Prim's algorithm uses a min-heap to determine the next edge with the least weight when building a minimum spanning tree. In the domain of data stream processing, heaps facilitate median maintenance, where two heaps (a min-heap and a max-heap) are maintained to dynamically compute the median as new elements arrive. Additionally, heaps are essential in job scheduling systems, where they help ensure that high-priority tasks are executed before lower-

priority ones, making them integral to high-performance and time-sensitive computing environments.

6.3.2 Characteristics and Properties of Heaps

1. Complete Binary Tree

- ◆ All levels are fully filled except possibly the last.
- ◆ The last level is filled from left to right, without gaps.

2. Heap Order Property

- ◆ All nodes must satisfy the heap condition relative to their parent-child relationship.

3. Efficient Indexing

- ◆ Heaps can be efficiently implemented using **arrays**, with relationships:
 - Left child of index i : $2i + 1$
 - Right child of index i : $2i + 2$
 - Parent of index i : $(i - 1) // 2$

4. Logarithmic Complexity

- ◆ Key operations (insertion, deletion, heapify) run in **$O(\log n)$** time.

6.3.3 Heap Property

The heap property is a fundamental rule that governs the organization of elements within a heap data structure. It ensures a specific relationship between parent and child nodes in a binary tree that allows for efficient access to the element of highest or lowest priority. The heap property comes in two main variants, depending on whether the heap is a min-heap or a max-heap.

◆ Min-Heap Property

In a min-heap, the heap property requires that the value of each parent node is less than or equal to the values of its children. This means the smallest element in the heap is always located at the root node. It does not require that the children themselves are in any particular order with respect to one another, only that they are not smaller than their parent.

Example (Min-Heap):

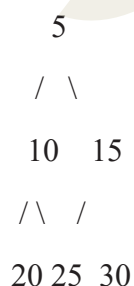


Fig: 6.3.1 Min-Heap

In this example, every parent node is less than or equal to its children, satisfying the min-heap property.

◆ Max-Heap Property

In a max-heap, the heap property states that the value of each parent node is greater than or equal to the values of its children. This ensures that the largest element is always present at the root node. Again, no specific order is required among siblings, but no child can be greater than its parent.

Example (Max-Heap):

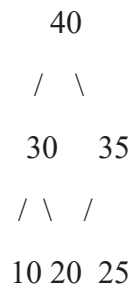


Fig:6.3.2 Max-Heap

Here, each parent has a value greater than or equal to its children, so the max-heap property holds.

The heap property is what allows the heap to function efficiently as a **priority queue**. Because the minimum or maximum value is always at the root:

- ◆ Accessing the highest or lowest priority element takes constant time ($O(1)$).
- ◆ Insertion and deletion operations can be performed in logarithmic time ($O(\log n)$) by adjusting the heap through a process known as heapify, which restores the heap property if violated.

◆ Heapify

Heapify is used to rearrange the elements to maintain heap property.

Max-Heapify Algorithm (Recursive):

```
def max_heapify(arr, n, i):  
    largest = i  
    left = 2*i + 1  
    right = 2*i + 2  
    if left < n and arr[left] > arr[largest]:  
        largest = left  
    if right < n and arr[right] > arr[largest]:  
        largest = right
```

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

max_heapify(arr, n, largest)

6.3.4 Representations of Heaps

A heap is a special kind of **binary tree** that satisfies a specific ordering property known as the heap property. In a **max heap**, the value of each parent node is greater than or equal to the values of its children. In a **min heap**, the value of each parent node is less than or equal to the values of its children. This makes heaps very useful in scenarios where quick access to the highest or lowest priority element is needed, such as in priority queues and scheduling algorithms.

Because heaps are binary trees, they can be represented in two main ways. The first method is the **linked representation**, where each node points to its left and right child. The second and more common method is the **array representation**, where the heap is stored in a linear array while maintaining the structure of a **complete binary tree**. In the array form, the position of the parent and children can be easily determined using index formulas. This representation simplifies memory usage and allows efficient implementation of heap operations.

6.3.4.1 Binary Heap Representation

- ◆ **Description:** A binary heap is the most common representation of a heap. It is a complete binary tree where each node satisfies the heap property: in a min-heap, every parent node is less than or equal to its children, while in a max-heap, every parent node is greater than or equal to its children. This property ensures that the root node contains the smallest (in a min-heap) or largest (in a max-heap) element.
- ◆ **Implementation:** Binary heaps are typically implemented using arrays, where the root of the tree is at index 0. For any element at index i , its left child is at index $2i + 1$ and its right child is at index $2i + 2$. The parent of an element at index i can be found at index $(i - 1) / 2$.

6.3.4.2 Fibonacci Heap Representation

- ◆ **Description:** A Fibonacci heap is a more complex heap data structure that supports a broader range of operations more efficiently than binary heaps. It consists of a collection of tree roots, where each tree is a min-heap. Fibonacci heaps provide amortized constant time for insertion and merging operations and logarithmic time for deletion and extraction of the minimum element.
- ◆ **Implementation:** Fibonacci heaps are implemented as a circular, doubly linked list of trees. Each tree in the heap maintains the heap property, and nodes can have multiple children, allowing for a more flexible structure compared to binary heaps. The structure supports efficient merging of heaps and decrease-key operations.

6.3.4.3 Binomial Heap Representation

- ◆ **Description:** A binomial heap is a collection of binomial trees, which are

a specific type of tree used to implement a priority queue efficiently. Each binomial tree in the heap follows the binomial tree property, where each tree in the heap is a binomial tree of a particular order.

- ◆ **Implementation:** Binomial heaps are implemented as a set of binomial trees, and operations such as insertion, union, and extraction of the minimum are performed by manipulating these trees. Each binomial tree is represented as a linked structure, allowing efficient merging of heaps.

6.3.4.4 Pairing Heap Representation

- ◆ **Description:** A pairing heap is a simpler alternative to Fibonacci heaps that supports efficient priority queue operations. It is a type of self-adjusting heap where nodes are organized in a tree-like structure, and the tree structure adjusts itself as operations are performed.
- ◆ **Implementation:** Pairing heaps are represented as a single tree or a collection of trees, where each node has a pointer to its children and a sibling pointer for efficient merging. The structure is designed to provide good performance for operations like insertion, deletion, and merging with a relatively simple implementation compared to Fibonacci heaps.

Each of these heap representations provides different trade-offs in terms of complexity, efficiency, and suitability for various applications, making them valuable tools in managing prioritized data.

6.3.5 Operations on Heap: Insertion and Deletion

The two most fundamental operations on a heap are insertion and deletion (specifically, deletion of the root element). Both operations maintain the heap property and the complete binary tree structure through a process called heapify.

1. Insertion into Heap

When a new element is inserted into a heap (min-heap or max-heap), it is initially placed at the end of the array (last level, leftmost vacant position) to maintain the complete binary tree property. Then, it is “heapified up” (also called “bubble up” or “percolate up”) until the heap order property is restored.

Algorithm (for Min-Heap Insertion):

1. Insert the element at the end of the heap.
2. Compare the inserted element with its parent.
3. If the inserted element is smaller than the parent, swap them.
4. Repeat step 2 until the inserted element is no longer smaller than the parent or it becomes the root.

Example:

Insert 5 into the following Min-Heap:

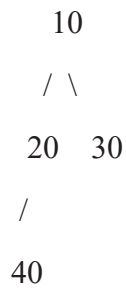


Fig 6.3.3 Min-Heap

Step 1: Insert 5 at the next available position:

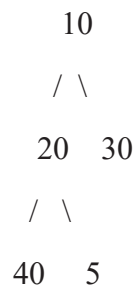


Fig 6.3.4 Insert 5 in Min-Heap

Step 2: Heapify Up

- ◆ $5 < 20 \rightarrow \text{swap}$
- ◆ $5 < 10 \rightarrow \text{swap}$

Resulting Heap:



Fig 6.3.5 Resulting-Heap

Insertion Algorithm (Max Heap)

1. Increase the size of the heap by 1.
2. Insert the new value at the end of the heap ($\text{heap}[\text{size}] \leftarrow \text{value}$).
3. Set $\text{current} \leftarrow \text{size}$
4. while $\text{current} > 1$ and $\text{heap}[\text{current}] > \text{heap}[\text{current} // 2]$ do
 swap $\text{heap}[\text{current}]$ and $\text{heap}[\text{current} // 2]$

current \leftarrow current // 2

5. end while

Step-by-Step Example

Initial Max Heap (array representation):

Index: 1 2 3 4 5

Heap: [50, 30, 40, 10, 20]

Insert value = 60

Step 1: Place 60 at the end

Heap: [50, 30, 40, 10, 20, 60]

Index: 1 2 3 4 5 6

Step 2: Heapify up

- ◆ 60 is at index 6 \rightarrow Parent is at index 3 (value = 40)
- ◆ $60 > 40 \rightarrow$ swap

Heap: [50, 30, 60, 10, 20, 40]

- ◆ Now 60 is at index 3 \rightarrow Parent is at index 1 (value = 50)
- ◆ $60 > 50 \rightarrow$ swap

Heap: [60, 30, 50, 10, 20, 40]

Now the max heap property is restored.

Final Max Heap (Tree Form)

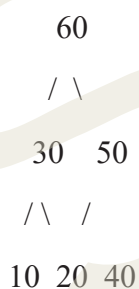


Fig:6.3.6 Final Min-Heap

2. Deletion from Heap

Deletion typically refers to removing the root element (min in min-heap or max in max-heap). To maintain the complete binary tree structure:

- ◆ Replace the root with the last element in the heap.
- ◆ Remove the last element.

- ◆ **Heapify down** (also called “**bubble down**” or “**percolate down**”) from the root to restore the heap property.

Algorithm (for Min-Heap Deletion):

1. Replace the root with the last element in the heap.
2. Remove the last element.
3. Compare the new root with its children.
4. If the root is greater than the smallest child, **swap** it with that child.
5. Repeat step 3 until the heap property is restored.

Example:

Delete the root (5) from the Min-Heap:

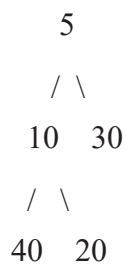


Fig 6.3.7 Min-Heap

Step 1: Replace 5 with the last element (20):

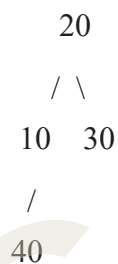


Fig 6.3.8 Min-Heap

Step 2: Heapify Down

$20 > 10 \rightarrow \text{swap}$

Resulting Heap:

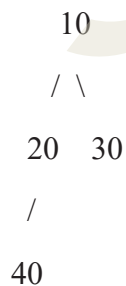


Fig 6.3.9 Resulting-Heap

Step-by-Step Example (Delete from Max Heap)

Initial Max Heap:

Index: 1 2 3 4 5 6

Heap: [60, 30, 50, 10, 20, 40]

Step 1: Remove the root (60), replace with last element (40)

Heap: [40, 30, 50, 10, 20]

Step 2: Heapify down from root

- ◆ Root: 40
- ◆ Children: 30 (left), 50 (right)
- ◆ Largest child is 50 → Swap 40 and 50

Heap: [50, 30, 40, 10, 20]

- ◆ Now at index 3: 40
- ◆ Children: none → Stop

Final Max Heap (Tree Representation):

```
      50
     /  \
    30   40
   /  \
  10  20
```

Fig 6.3.10 Final Max-Heap

6.3.6 Advantages and Disadvantages of Heaps

The heap data structure offers a range of advantages, especially in contexts where managing elements by priority is essential. One of the most notable strengths of heaps lies in their ability to efficiently perform core operations like insertion, deletion, and extracting the minimum or maximum element, depending on whether it is a min-heap or max-heap. These operations are executed in $O(\log n)$ time in a binary heap, which is significantly faster than linear time required in data structures like unsorted arrays. Heaps are particularly well-suited for implementing priority queues, where tasks must be handled based on urgency rather than order of arrival. They also serve as the foundational structure for efficient algorithms such as Heapsort, which utilizes the heap property to sort data with good time complexity and in-place sorting. Moreover, because binary heaps are typically implemented using arrays, they offer simple memory management and constant-time access to parent or child nodes through index arithmetic, streamlining both implementation and performance.

Despite their many advantages, heaps come with certain limitations. A key drawback is the lack of efficient access to arbitrary elements other than the root. Retrieving or modifying elements deeper in the heap requires linear time, making it unsuitable for applications where frequent searches or random access are necessary. This contrasts with structures like binary search trees, which offer better average-case access times for a range of elements. Additionally, while binary heaps are efficient for standard priority operations, they are not optimal for more advanced tasks such as the decrease-key operation, which plays a critical role in graph algorithms like Dijkstra's algorithm. In such cases, more complex structures like Fibonacci heaps outperform binary heaps due to their superior amortized time complexities, albeit at the cost of increased implementation complexity. Therefore, while heaps are highly effective in many use cases, selecting the right heap variant depends on the specific performance requirements and operational constraints of the problem being solved.

Recap

- ◆ A heap is a complete binary tree that satisfies the heap property, making it efficient for priority-based operations.
- ◆ Heaps are widely used in applications such as priority queues, operating system scheduling, graph algorithms, and Heapsort.
- ◆ The heap property ensures that in a min-heap, the parent is less than or equal to its children; in a max-heap, the parent is greater than or equal to its children.
- ◆ Heaps are implemented using arrays, enabling efficient indexing with simple formulas for parent and child relationships.
- ◆ The root of a heap always holds the element with the highest or lowest priority, depending on the heap type.
- ◆ Heapify is the process used to maintain the heap property after insertion or deletion.
- ◆ Insertion involves placing a new element at the end and performing heapify up to restore order.
- ◆ Deletion typically removes the root element, replaces it with the last element, and uses heapify down to restore the heap structure.
- ◆ Binary heaps are the most common, while Fibonacci, binomial, and pairing heaps offer alternative structures for specific use cases.
- ◆ Heaps provide logarithmic time complexity for insertion and deletion and constant time for accessing the root.
- ◆ They are memory-efficient when implemented as arrays and support in-place sorting with Heapsort.

- ◆ Heaps are not suitable for fast access to arbitrary elements and perform poorly in search operations compared to binary search trees.
- ◆ Advanced operations like decrease-key are inefficient in binary heaps and better handled by Fibonacci heaps.
- ◆ Choosing the right type of heap depends on the application's performance requirements and operation types.

Objective Type Questions

1. Name the data structure that supports priority-based scheduling.
2. In a min-heap, where is the smallest element always located?
3. State the time complexity of insertion in a binary heap.
4. Identify the heap type used in Dijkstra's algorithm.
5. Give the term for the process used to maintain the heap property.
6. Mention the position of the left child of a node at index i in an array-based heap.
7. Name the most basic and commonly used heap representation.
8. Accessing the root element of a heap takes how much time?
9. Which heap is self-adjusting and easier to implement than Fibonacci heaps?
10. Point out the traversal method used to store a heap in an array.
11. State the primary advantage of using heaps in real-time systems.
12. Name the process of moving an inserted element upward to restore the heap.
13. Which heap has the best amortized time for decrease-key operation?
14. What is the root index in a heap stored as an array?
15. Which sorting algorithm is based on heap data structure?
16. What kind of tree is a heap?
17. What is the time complexity of deleting the root from a heap?

Answers to Objective Type Questions

1. Heap
2. Root
3. $O(\log n)$
4. Min-Heap
5. Heapify
6. $2i+1$
7. Binary
8. $O(1)$
9. Pairing
10. Level-order
11. Priority
12. Bubble-up
13. Fibonacci
14. 0
15. Heapsort
16. Complete
17. $O(\log n)$

Assignments

1. Explain the structure, properties, and applications of the heap data structure in detail.
2. Differentiate between min-heap and max-heap with suitable examples and diagrams.
3. Describe the heapify operation and its role in insertion and deletion in a binary heap.

4. Discuss various types of heaps including binary heap, Fibonacci heap, binomial heap, and pairing heap.
5. Write and explain the algorithm for inserting a new element into a max-heap with an example.
6. Compare the advantages and disadvantages of using a heap data structure.

Reference

1. Rao, G. A. V. (2021). *Data structures and algorithms using C and C++*. PHI Learning.
2. Narayan, S. (2021). *Data structures and algorithms made easy: Data structure and algorithmic puzzles* (5th ed.). CareerMonk Publications.
3. Narasimha Karumanchi. (2021). *Data Structures and Algorithms Made Easy* (7th ed.). CareerMonk Publications..
4. E. Balagurusamy. (2020). *Fundamentals of Data Structures Using C*. McGraw-Hill Education.
5. Lipschutz, S. (2014). *Data Structures (Schaum's Outlines Series)*. McGraw-Hill Education.

Suggested Reading

1. Yashavant Kanetkar. (2022). *Data Structures Through C*. BPB Publications.
2. A. M. Tenenbaum, Y. Langsam & M. J. Augenstein. (2022). *Data Structures Using C*. Pearson Education India.
3. Lipschutz, S. (2014). *Data structures* (Schaum's outlines series, Revised ed.). McGraw-Hill Education.
4. Kruse, R. L., Leung, B. P., & Tondo, C. L. (1999). *Data structures and program design in C* (2nd ed.). Pearson Education.

Unit 4

Priority Queues

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- ♦ define a priority queue and explain how it differs from a regular queue.
- ♦ familiarise the types of priority queues and their characteristics.
- ♦ identify the key operations performed on a priority queue.
- ♦ explore various methods used to implement a priority queue.

Prerequisites

Imagine a hospital emergency room where multiple patients arrive at the same time - some with minor injuries and others in critical condition. Who gets treated first? Clearly, not just the one who arrived first. The decision depends on priority.

Now, think of systems like Google Maps, CPU schedulers, or airline check-in counters - all of them handle tasks that cannot simply wait in line based on arrival time alone. They need a smarter way to decide who goes next.

In this lesson, you are about to uncover the powerful logic behind such decision-making. Get ready to explore the idea that powers smart queues, where importance matters more than order.

Key words

Priority queue, max priority, min priority, heapify, min-heap, max-heap

Discussion

Imagine you're at a crowded airport check-in counter. Passengers are lined up, waiting to drop off their luggage and collect boarding passes.

There are two different queues:

In the regular queue, passengers are helped in the order they arrive. It is strictly first come, first served. In the VIP queue, passengers are served based on importance or urgency, even if they showed up later than others.



This VIP line is the perfect way to understand a Priority Queue — a special kind of data structure that puts priority first!

ie, A priority queue is an advanced data structure that extends the concept of a traditional queue by associating a priority with each element. They are abstract data type, which means they are defined by the operations they support rather than the way they are implemented. Unlike a regular queue where elements are processed in the order they arrive (FIFO – First In, First Out), a priority queue ensures that the element with the highest priority is processed first, regardless of its insertion order.

What Makes It Special?

In a priority queue, every person (or data element) has a priority number. The lower the number, the higher the priority (in a min-priority queue). So, someone with priority 1 gets ahead of someone with priority 3, even if they joined the line later.

Just like:

- ◆ A mother with an infant (priority 1)
- ◆ An elderly person (priority 2)
- ◆ A business-class passenger (priority 3)
- ◆ A regular tourist (priority 5)

6.4.1 Types of Priority Queues

Min Priority Queue

In this type of priority queue, the smaller the value, the more important it is. So, elements with lower numbers get served first. Think of it like a queue where the quietest voices are heard first. For example, if the queue contains the numbers 7, 3, 5, 9, and 2, the number **2** will be the first to come out because it's the smallest and thus has the highest priority.

Max Priority Queue

Here, it's the opposite. Higher values are considered more important. So the bigger the number, the sooner it gets removed from the queue. Imagine you're picking the tallest players for a basketball team, taller players (higher numbers) go first. If you add 11, 24, 6, 19, and 30 to this queue, **30** will be the first one dequeued because it's the largest and holds the highest priority.

6.4.2 Implementation of Priority Queue

Priority queues can be implemented using a variety of data structures, each offering different performance characteristics. The most common implementation is with a binary heap, which provides efficient insertion and deletion operations. Other possible implementations include binary search trees, arrays, and linked lists, depending on the specific use case and performance requirements. Regardless of the underlying structure, a priority queue supports three main operations: insertion, where an element is added along with an associated priority; deletion (often referred to as dequeue), which removes and returns the element with the highest priority; and peek, which allows viewing the

highest priority element without removing it. These operations are essential in scenarios where elements must be processed based on their importance rather than the order in which they arrive.

6.4.2.1 Implementation of Priority Queue using Array

As already discussed, Priority queue in a data structure is an extension of a linear queue that possesses the following properties:

- ◆ Priority Assignment: Each element has a defined priority.
- ◆ Comparability: All elements must be comparable so their priorities can be ordered.
- ◆ Removal Order: Elements with higher priority are removed before those with lower priority.
- ◆ Tie Handling: If priorities are equal, elements follow the regular queue rule (FIFO).

Let us understand these properties through an example. Suppose you need to insert the elements 7, 2, 45, 32, and 12 into a priority queue. In this case, elements with smaller values are considered to have higher priority. Therefore, the queue should be arranged in such a way that the smallest element is positioned at the front.

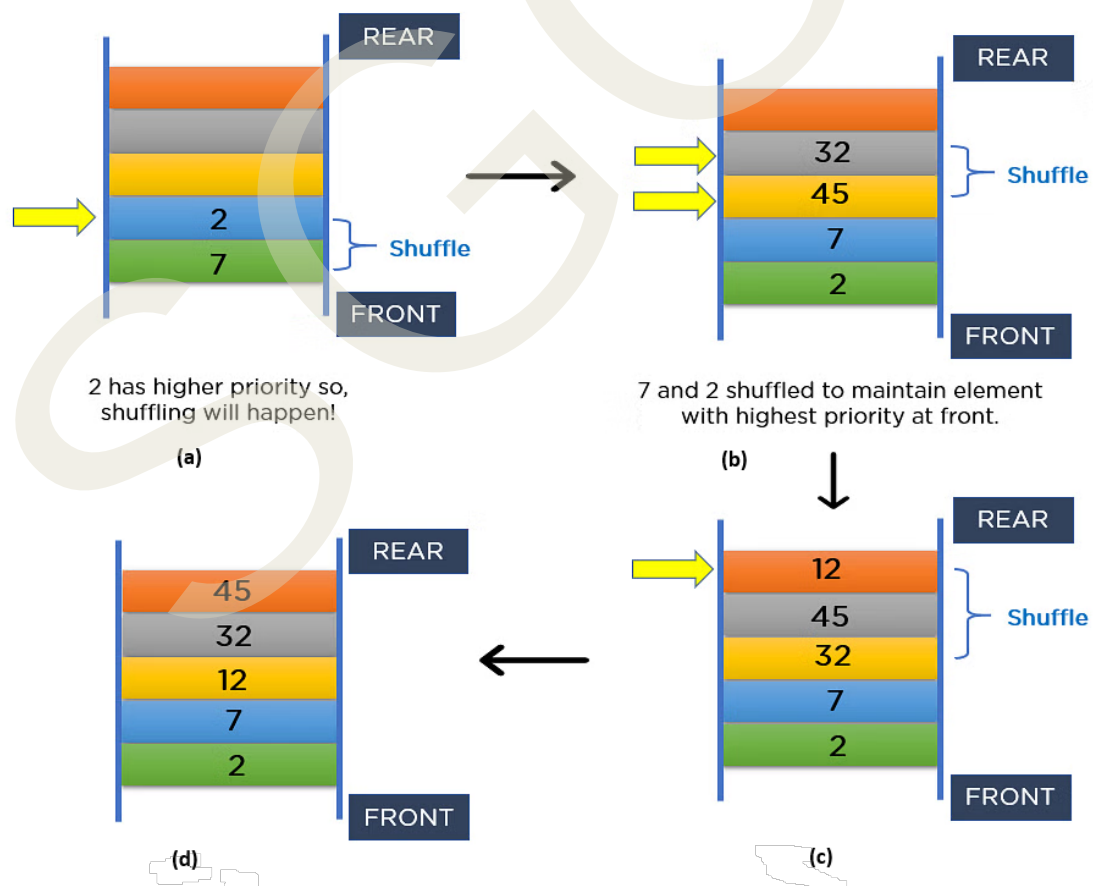


Fig 6.4.1 (a), (b),(c), (d) Implementation of priority queue using array

Fig. 5.4.1 illustrates how a priority queue maintains element order during insertion. The steps involved are as follows:

1. Enqueue 7: Insert 7 into the queue.
2. Enqueue 2: Since 2 has higher priority (i.e., a lower value), it is placed before 7. The queue is reordered accordingly, as shown in Fig. 5.4.1(a).
3. Enqueue 45: Insert 45 at the appropriate position based on its priority.
4. Enqueue 32: As 32 has higher priority than 45, swap them to maintain the priority order. See Fig. 5.4.1(b).
5. Enqueue 12: Reorder 12, 32, and 45 based on priority. The updated order is shown in Fig. 5.4.1(c).

The final queue arrangement is depicted in Fig. 5.4.1(d), where elements are sorted based on their priority. Now, since elements are arranged in priority order (lowest value = highest priority), dequeue operations can be efficiently performed from the rear end. However, if we perform n comparisons during each insertion to maintain this order, the overall time complexity becomes $O(n^2)$, which is inefficient for large data sets.

6.4.2.2 Implementation of Priority Queue using Linked list

When implementing a priority queue using an array, inserting elements into a sorted array takes $O(n)$ time. As a result, processing each element can lead to an overall time complexity of $O(n^2)$. Due to this inefficiency, arrays are not considered an ideal choice for implementing priority queues. Therefore, in this context, we will focus on representing priority queues using a linked list, which offers a more suitable alternative.

To implement a priority queue using a linked list, the first step is to create a node structure where each node stores three things: the data (value), its priority (which determines its importance), and a pointer to the next node in the list. Initially, the linked list is empty, and a pointer (commonly named front) is used to track the beginning of the queue.

When inserting a new element, it must be placed in the list based on its priority. Higher-priority elements are inserted before lower-priority ones. This involves traversing the list to find the correct position for the new node. To delete an element, simply remove the front node since it holds the highest priority. To peek at the highest-priority element, return the data of the front node without removing it. These operations such as insert, delete, and peek can be repeated as needed to manage the queue efficiently.

Consider a linked queue having four data elements 10, 4, 30, 6, and 25. We have to insert it into a priority queue.

Let's assume:

The priority is determined by lower value = higher priority (i.e., the smaller the number, the sooner it should be dequeued). We're using a linked list to implement the priority queue.

Step 1: Insert 10

The queue is empty. So, 10 becomes the first node (head).

Queue: 10

Step 2: Insert 4

4 has higher priority than 10 (since $4 < 10$), so insert it at the front.

Queue: $4 \rightarrow 10$

Step 3: Insert 30

30 has the lowest priority so far.

Traverse the list until the end and insert it at the last.

Queue: $4 \rightarrow 10 \rightarrow 30$

Step 4: Insert 6

6 is greater than 4 but less than 10. So insert between 4 and 10.

Queue: $4 \rightarrow 6 \rightarrow 10 \rightarrow 30$

Step 5: Insert 25

25 is greater than 10 but less than 30. Insert it between 10 and 30.

Queue: $4 \rightarrow 6 \rightarrow 10 \rightarrow 25 \rightarrow 30$

6.4.2.3 Implementation of Priority Queue using Heap

A heap is a special tree-like structure that is always a complete binary tree. It follows a rule called the heap property. There are two types of heap data structures: Max Heap and Min Heap. In a Max Heap, the value of each parent node is greater than or equal to the values of its child nodes. It follows a tree-like structure where the largest value is always at the root. The diagram below illustrates a binary max heap with the maximum value positioned at the root node.

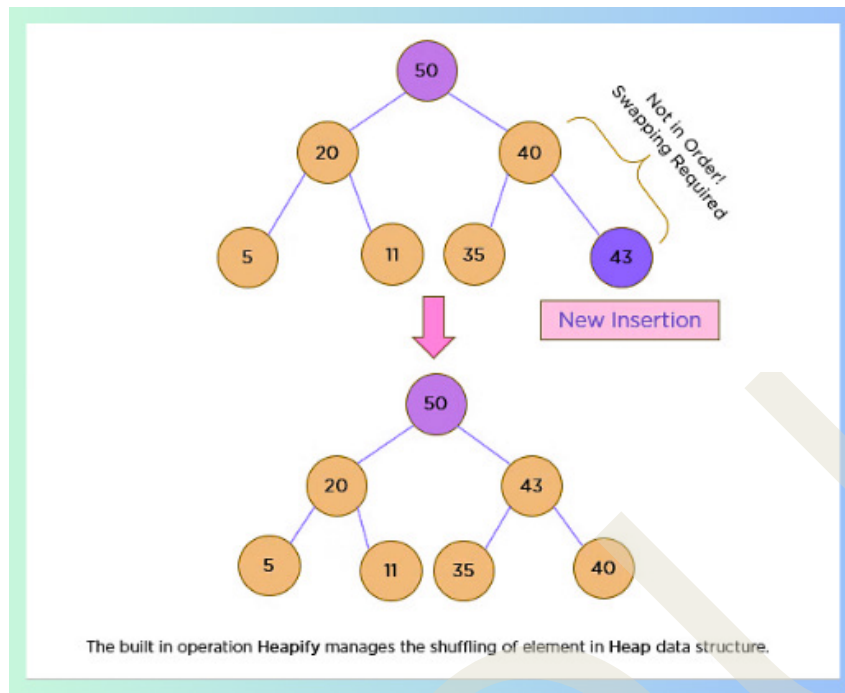
The main operations you can perform on a priority queue are insertion, deletion, and peek. Let's see how these work in a max heap representation of a priority queue.

1. Inserting an Element into a Priority Queue

When you insert a new element, it is first placed in the next available position from top to bottom and left to right (just like filling a binary tree). After placing it, the element is compared with its parent.

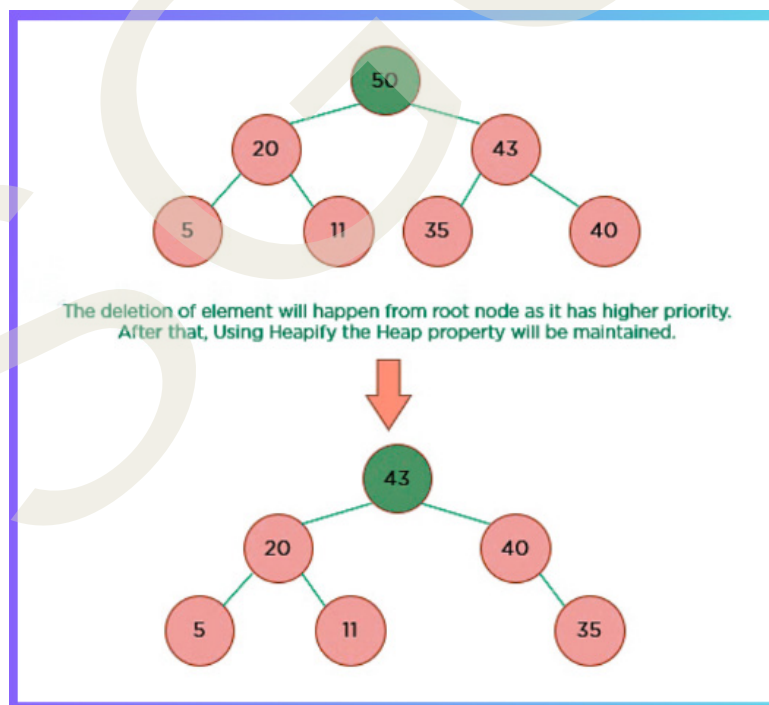
- ◆ If the new element is greater than its parent, they are **swapped**.
- ◆ This process continues. The element keeps moving up (swapping) until it reaches the correct position in the heap.

In the above example, the new element 43 was inserted into the max heap representation of the priority queue. However, this insertion disrupted the heap order. To restore the correct order, a series of swapping operations are performed. This process is known as heapify (which you have studied in the previous unit) in the heap data structure. It ensures that the max heap property is maintained after insertion.



2. Deleting an element from Priority queue

As you know that in a max heap, the maximum element is the root node. This removal leaves an empty slot at the root, which is then filled by another element. Usually it is the last element in the heap. To maintain the heap property, the inserted element is compared with its children and appropriately repositioned within the heap structure.



In the example above, it will delete the element at the root node, and the heapify operation is performed to restore the priority-based order.

Recap

Definition

- ◆ A **priority queue** is an abstract data type in which each element is associated with a priority.
- ◆ Unlike a regular queue (FIFO), elements are served based on their priority.
- ◆ The element with the **highest priority** is dequeued first.

Types of Priority Queues

- ◆ **Min Priority Queue:** Lower priority value means higher priority.
- ◆ **Max Priority Queue:** Higher priority value means higher priority.

Key Properties

- ◆ Each element has an associated priority.
- ◆ Elements are not always removed in the order they are inserted.
- ◆ If two elements have the same priority, they are dequeued in insertion order (FIFO).
- ◆ Priority queues are commonly used in scheduling, pathfinding, and resource management.

Implementation Methods

1. Array

- ◆ Maintains order based on priority.
- ◆ Inefficient for large data due to frequent rearrangements.
- ◆ Time complexity: Insertion and deletion may take $O(n)$ time.

2. Linked List

- ◆ Nodes contain data and priority.
- ◆ Requires traversal to insert based on priority.
- ◆ Insertion and deletion times vary ($O(n)$ for insertion, $O(1)$ for deletion from front).

3. Heap (Preferred for Efficiency)

- ◆ A complete binary tree that maintains the heap property.
- ◆ **Max Heap:** Parent nodes are greater than or equal to child nodes.

- ◆ Operations:
 - **Insertion:** Insert at the end and restore order using “heapify up”.
 - **Deletion:** Remove the root, replace with the last node, and restore order using “heapify down”.
- ◆ Time complexity: $O(\log n)$ for both insertion and deletion.

Basic Operations

- ◆ **Insert:** Add an element with a given priority.
- ◆ **Delete:** Remove the highest-priority element.
- ◆ **Peek:** View the highest-priority element without removing it.

Objective Type Questions

1. What type of data structure serves elements based on priority?
2. Which priority queue removes the element with the smallest value first?
3. What is the most efficient data structure used to implement a priority queue?
4. What operation retrieves the highest-priority element without removing it?
5. In which queue type does a higher number indicate higher priority?
6. What is the name of the tree structure commonly used in heaps?
7. Which operation maintains the heap property after insertion?
8. What is the operation used to maintain order after deletion in a heap?
9. What keyword refers to arranging a tree to maintain heap property?
10. What structure is used in linked list implementation of a priority queue?

Answers to Objective Type Questions

1. PriorityQueue
2. MinHeap
3. Heap

4. Peek
5. MaxHeap
6. BinaryTree
7. HeapifyUp
8. HeapifyDown
9. Heapify
10. Node

Assignments

1. Explain the concept of a priority queue. Discuss its types and how it differs from a regular queue. Give suitable examples to support your explanation.
2. Describe different methods of implementing a priority queue using arrays, linked lists, and heaps. Compare their efficiency in terms of time complexity.
3. What is a heap? Explain how it is used in the implementation of a priority queue. Discuss the operations involved, such as insertion and deletion, with proper examples.
4. Distinguish between min-priority queue and max-priority queue. Explain their working mechanism and provide examples to show how elements are added and removed.
5. Write and explain the step-by-step algorithm to insert and delete elements in a max-priority queue using a binary heap. Use diagrams or sample data to illustrate the process.
6. Discuss the applications of priority queues in real-world scenarios such as CPU scheduling, Dijkstra's algorithm, or job scheduling systems. Explain the importance of priority in each case.
7. Define heapify in the context of heaps used in priority queues. Explain the difference between heapify-up and heapify-down with the help of examples.

Reference

1. Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
2. Horowitz, E., Sahni, S., & Mehta, D. (2008). *Fundamentals of Data Structures in C++* (2nd ed.). Universities Press.
3. Lafore, R. (2002). *Data Structures and Algorithms in C++* (2nd ed.). Sams Publishing.

Suggested Reading

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). *Data Structures and Algorithms in C++* (2nd ed.). Wiley.
3. Malik, D. S. (2010). *Data Structures Using C++* (2nd ed.). Cengage Learning.



QP CODE:

Reg. No :

Name :

Model Question Paper- set-I

End Semester Examination

BSc. Data Science and Analytics

B24DS05DC:DATA STRUCTURES

(CBCS - UG)

2024-25 - Admission Onwards

Time: 2 Hours

Max Marks: 45

Section A

Answer any 10 questions. Each carries one mark

(10 x 1 = 10)

1. What keyword is used to define a constant in C?
2. Which of the following is a non-primitive data structure?
 - a. Char
 - b. Float
 - c. Queue
 - d. Boolean
3. Name one application of a linked list.
4. A binary tree in which every node has 0 or 2 children?
5. What is heap?
6. Write the classifications of non-primitive data types.
7. Which statement allows multi-way branching?
8. What is the index of the first element in an array?
9. Name two basic operations performed on a linked list.

10. What is the main data structure used in hashing?
11. What is trie data structure?
12. List different types of Queue.
13. What keyword is used to define a structure in C?
14. Which collision method uses a second hash function?
15. Which type of linked list has its last node pointing back to the first node?

Section B

Answer any 5 questions. Each carries two marks

(5 x 2 = 10)

16. What is Dynamic Memory Allocation ?
17. What is a one-dimensional array?
18. What is meant by traversal in a linked list?
19. What is an undirected graph?
20. Explain any two Properties of Hash Functions.
21. Explain the advantages of Red Black Tree?
22. What is a pointer to an array?
23. Explain any two applications of stack in briefly.
24. What is a self-referential structure?
25. What is the degree of a vertex in a graph?

Section C

Answer any 5 questions. Each carries four marks

(5 x 4 = 20)

26. Explain the different representations of heap.
27. Explain the process of balancing a Red-Black Tree after deletion with a suitable case.
28. Differentiate between entry-controlled and exit-controlled loops.
29. Define a linear data structure with an example.

30. Explain the representation and basic operations of a stack using a linked list.
31. Draw a binary tree of height 3 and label root, internal, and leaf nodes.
32. Distinguish between min-priority queue and max-priority queue
33. Describe a circular queue.
34. Explain any 4 operators used in C
35. Explain any two primary operations of a stack.

Section D

Answer any 2 questions. Each carries fifteen mark (2 x 15 = 30)

36. Explain the heap data structure in detail ? Differentiate between min-heap and max-heap with suitable examples and diagrams.
37. Write a C program to define a structure called Student with members: roll_no, name, and marks. Create an array of 5 students, accept details from the user, and display the student with the highest marks.
38. Explain the Queue data structure in detail. Include its basic operations, and explain how the FIFO principle works with an example.
39. Explain different collision resolution techniques in hashing and explain its advantages and disadvantages.



SREENARAYANAGURU OPEN UNIVERSITY

QP CODE:

Reg. No :

Name :

Model Question Paper- set-I

End Semester Examination

BSc. Data Science and Analytics

B24DS05DC:DATA STRUCTURES

(CBCS - UG)

2024-25 - Admission Onwards

Time: 2 Hours

Max Marks: 45

Section A

Answer any 10 questions. Each carries one mark (10 x 1 = 10)

1. The expression $a \ ? \ b \ : \ c$ is an example of which operator?
2. What does LIFO stand for?
3. Name the data structure used to represent a queue using a linked list.
4. Binary tree filled from left to right is?
5. What is Hashing?
6. What is the use of peek operation in stack?
7. In a doubly linked list, which pointer points to the previous node?
8. What type of queue allows insertion and deletion from both ends?
9. What is the pointer called that refers to the first node in a linked list?
10. Traversal visiting nodes in Left-Root-Right order?
11. Which collision resolution technique uses a linked list at each index of a hash table?
12. Which type of data structure is Queue?



13. In a linked stack, which operation adds an element?
14. What is the total number of elements in a 2D array declared as `int arr[4][5];`?
15. Identify the data structure where insertion happens at one end and deletion happens at the other.

Section B

Answer any 5 questions. Each carries two marks

(5 x 2 = 10)

16. What are Nested loops?
17. Differentiate between array and linked list.
18. Name two types of linked lists and state one feature of each.
19. Define a tree and explain its basic terminology.
20. Write any two applications of trie.
21. Write about enqueue and dequeue operations.
22. Mention any two advantages of functions.
23. What is the difference between static and dynamic memory allocation?
24. Mention two advantages of using a circular linked list.
25. What is an AVL tree?

Section C

Answer any 5 questions. Each carries four marks

(5 x 4 = 20)

26. Explain advantages and disadvantages of heap.
27. Write about any two advantages and disadvantages of heap.
28. Differentiate Static and Dynamic memory.
29. Describe the Tower of Hanoi problem and explain how stacks can be used to solve it.
30. Define B-tree. Explain its basic properties.
31. Define a graph and explain types of graphs with examples.
32. Explain the different methods used to represent priority queues.

33. What is a Circular linked list? Explain the basic operations of a circular linked list.
34. Why break, continue and goto statements are used?
35. Write a short note on application of queues.

Section D

Answer any 2 questions. Each carries fifteen mark (2 x 15 = 30)

36. Draw a binary tree of height 4 and show all traversals.
37. Discuss advantages of BST over a linear data structure like linked list.
38. Explain basic and advanced data structures, and compare their differences.
39. Explain about the implementation of priority queues using array, linked list and heap.

സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുറിപ്പ് ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode

Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery

Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura

Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi

Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

**DON'T LET IT
BE TOO LATE**

**SAY
NO
TO
DRUGS**

**LOVE YOURSELF
AND ALWAYS BE
HEALTHY**



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala



Data Structures

COURSE CODE: B24DS05DC

SGOU



YouTube



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-994027-6-8



9 788199 402768